ENABLING EDGE-INTELLIGENCE IN RESOURCE-CONSTRAINED AUTONOMOUS SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Aqeel Anwar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Georgia Institute of Technology
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2021

ENABLING EDGE-INTELLIGENCE IN RESOURCE-CONSTRAINED AUTONOMOUS SYSTEMS

Thesis committee:

Dr. Arijit Raychowdhury (Advisor)
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
College of Computing
School of Computer Science
Georgia Institute of Technology

Dr. Justin Romberg
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Titash Rakhsit
Director
Qualcomm

Dr. Muhannad Bakir
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date approved: June 15, 2021

وَوَجَدَكَ ضَآلًّا فَهَدَىٰ

Wa Wajadaka Dāllāan Fahadá

And He found you lost and guided [you]

Quran [93:7]

To my beloved parents,

my supportive siblings,

Adeel and *Israa*,

my lovely wife

Sahrish,

and

مِنْحَة

(blessing of Allah)

for their

unconditional love,

support, and guidance

throughout the pursuit of

this and many other journeys.

ACKNOWLEDGMENTS

I would like to begin by thanking my Ph.D. research advisor *Dr. Arijit Raychowdhury* for his continuous support and patience throughout the last four years. Without his guidance, I would not have been able to overcome the difficulties in the course of research. He has been a constant source of advice for both my professional and personal life. A Ph.D. graduate is a reflection of his advisor, and I believe these years have positively impacted me making me a better person.

I would also like to express gratitude to my thesis committee members - *Dr. Justin Romberg, Dr. Muhannand Bakir, Dr. Titash Rakshit*, and *Dr. Hyesoon Kim* for providing me with extensive professional insights into my research. Their valuable suggestions went a long way in improving the quality of this document. I would also like to thank the members of my research group, *Dr. Saad, Dr. Samantak, Dr. Abhinav, Dr. Insik, Dr. Muya, Dr. Ningyuan, Dr. Anvesha, Dr. Jong-Hyeok, Dr. Yan, Rakshith, Bitan, Anupam, Foroozan, Aswhin, Brian, Sam*, and *Zishen*, who shaped my experience as a graduate student.

I would like to thank *Insik*, who helped me in my early Ph.D. years through his keen observations, revealing the secrets of Ph.D. life and how to cope up with it. M*uya* has constantly been a go-to person for me, whether I locked myself out of my lab, needed some random screwdriver, or some fancy item for my work desk. Sharing my lab space with *Jong-Hyeok* has been a great learning experience. I haven't seen such a humble and organized person during my stay here at Georgia Tech. My collaboration with *Sihan* was a great learning experience for me. Finally, I would like to mention *Zishen*, who I really enjoyed working with within my last year of Ph.D.

I would also like to thank two other people whom I have worked with in the past. I have had the opportunity of working with *Dr. Titash* as an intern at Samsung Semiconductor. Working with him taught me a sense of responsibility and ownership. He showed me how a good manager can increase your productivity and interest in your task. I worked with

Dr. Tahir, for my undergraduate senior year project, and later as a research assistant. He was the one who convinced me to apply for graduate studies in the US. He has been a great mentor providing me with perspective and direction to my career when I needed it the most.

The *Pakistani Community* here at Georgia Tech has been a vital part of my Ph.D. journey and has provided me with a home away from home. The Pakistan House in Homepark, the Eid gatherings, the cricket tournaments, the yearly farewells, and the adventurous trips with the friends I made, will always be a cherished part of my memory.

Finally, nobody has been more supportive in the pursuit of this journey than my family members. I would like to thank my parents, *Muhammad Anwar* and *Nusrat Jabeen*, who have believed in me, my brother, *Adeel*, who has always been there for me and provided me with the best of advice, my sister *Israa* who has always been very supportive, my cousin-sisters *Hooria, Areeba* and *Adeeha* whose love and prayers has always been with me. Last but not the least, I can't thank enough to my amazing wife, *Sahrish*. She would often help me debug my ideas, troubleshoot my algorithms, and even fix the drone when it would not fly. If it were not for her, I would have taken longer to complete my Ph.D.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this research is to shift Machine Learning algorithms from resource-extensive server/cloud to compute-limited edge nodes by designing energy-efficient ML systems. Multiple sub-areas of research in this domain are explored for the application of drone autonomous navigation. Our principal goal is to enable the UAV to autonomously navigate using Reinforcement Learning, without incurring any additional hardware or sensor cost. Most of the light-weight UAVs are limited in their resources such as compute capabilities and on-board energy source, and the conventional state-of-the-art ML algorithms can not be directly implemented on them. This research addresses this issue by devising energy-efficient ML algorithms, modifying existing ML algorithms, designing energy-efficient ML accelerators, and leveraging the hardware-algorithm co-design.

RL is notorious for being data-hungry and requires trials and error for it to converge. Hence it can not be directly implemented on real drones until the issues of safety, data limitations, and reward generation is addressed. Instead of learning the task from scratch, just like humans, RL algorithms can benefit from prior knowledge which can help them converge to their goals in less time and consume less energy. Multiple drones can be collectively used to help each other by sharing their locally learned knowledge. Such distributive systems can help agents learn their respective local tasks faster but may become vulnerable to attacks in the presence of adversarial agents which needs to be addressed.

Finally, the improvement in energy efficiency of RL-based systems achieved from the algorithmic approaches is limited by the underlying hardware and compute architectures. Hence, these need to be redesigned in an application-specific way exploring and exploiting the nature of the most commonly used ML operators This can be done by exploring new compute devices and taking into account the data reuse and dataflow of ML operators within the architectural design.

This research discusses these issues by addressing them and presenting better alterna-

tives. It is concluded that energy consumption at multiple levels of hierarchy needs to be addressed by exploring algorithmic, hardware-based, and algorithm-hardware co-design approaches.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Machine Learning (ML) algorithms and Deep Neural Networks (DNNs) are used for a wide number of varied applications such as computer vision, image and speech recognition, [1, 2], playing games [3], autonomous driving [4], robotics [5] and disease detection [6]. ML algorithms date back to the early 1980s, with significant contributions seen from the research community in the last decade. The success of ML is based on its ability to extract useful high-level features from the provided data. Each layer in the DNN acts as a feature extractor operating on the output of the previous layer. As we go deep into the network the features extracted become more and more problem-specific. Deeper DNNs have proved to perform better as compared to shallower ones for complex problems providing state-of-the-art accuracy even surpassing humans in some tasks. This superior accuracy, however, comes at the cost of high computational complexity. The deeper a DNN is, the more energy and time is required to train it and hence the accuracy-energy trade-off.

Recently there has been an increase in the use of IoT devices. According to a recent survey, the number of IoT devices surpassed 50 billion in the year 2020 [7]. These IoT devices are limited in terms of their resources such as battery and compute capabilities. Most of the known state-of-the-art ML algorithms cannot be directly implemented on them due to the large energy and compute gap. For Supervised and Unsupervised machine learning problems, where the training data is available apriori, this problem is not that worse. In general, training a DNN takes up more computational resources than using it in the inference mode. So for these problems, where there is a clear boundary between the training and inference phase, the DNNs can be trained offline on the cloud which has plenty of

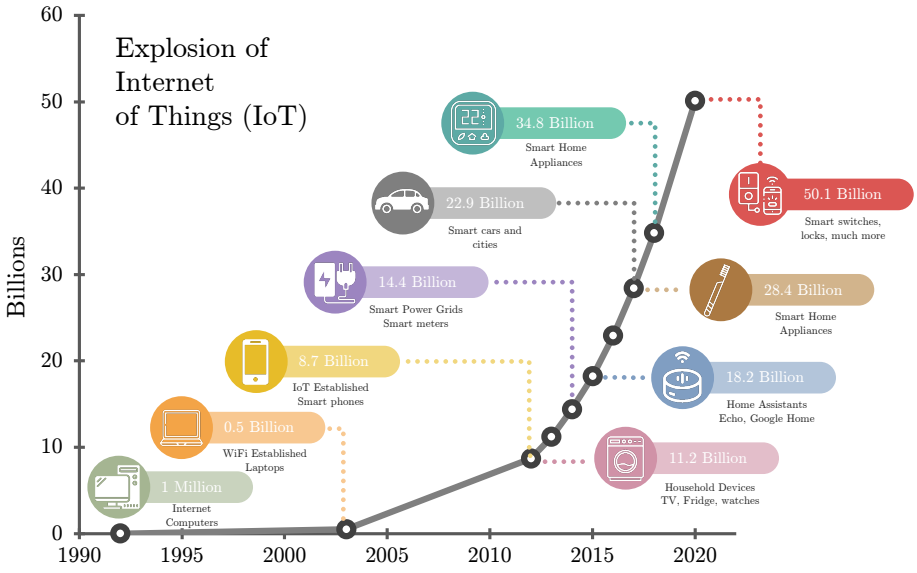


Figure 1.1: Number of IoT devices over the years

computing resources at its disposal. Once the DNN is trained, it can be transferred to the IoT device which then might be able to use it in the inference phase. The problem is worse for Reinforcement Learning (RL) based applications. In RL, the learning agent interacts with the underlying environment to generate data which is then used for training. Due to the unavailability of the training data pairs apriori, the training needs to happen at the edge node. Since these edge nodes are resource-constrained, implementing DNN training on them becomes nearly impossible. Hence energy efficient ML systems need to be designed.

1.2 Prior Work

In the past few years, researchers have started to realize the importance of energy efficiency in ML algorithms. A lot of people have started working towards making the existing ML algorithms energy and latency efficient. Instead of using energy-independent Key Performance Indices (KPIs), researchers have started using energy-aware KPIs such as performance per watt. The work done by researchers in this area can be categorized into three main approaches and has been summarized in Figure 1.2.

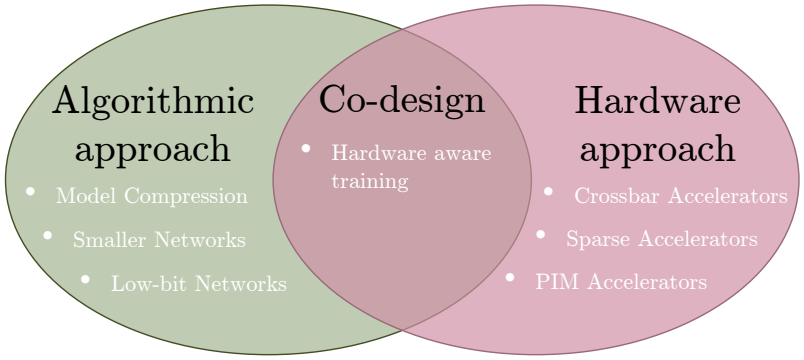


Figure 1.2: Different approaches to designing energy efficient ML systems

1.2.1 Algorithmic Approach

The first approach is to address the problem from a purely algorithmic standpoint. This involves directly reducing the floating-point operations (FLOPs) required to traverse the neural network without losing accuracy. Researchers have worked on implementing small and efficient neural networks with similar accuracy. Among the two broad categories, one category is the Model compression technique [8, 9, 10, 11, 12]. One of the most common model compression technique is pruning where the network is scanned for unnecessary or redundant connections [12, 13, 14]. In [11], SVD decomposition is carried out on the weight matrices of a pre-trained DNN model to get rid of redundant dimensions while preserving the model accuracy. Researchers in [12] use network pruning on a pre-trained DNN model and then replaces the weights of DNN which are below a threshold with zero. The network is trained again to regain the accuracy. This process is carried out multiple times until there is no room for pruning. Both of these approaches require the DNN to be trained multiple times which in turn means more compute energy. Another approach to implementing model compression is using Quantization where the precision or the width of the network weights is reduced [15, 16, 17, 18, 19]. Quantized neural networks are introduced in [18], where the weights of the DNN layers are quantized to low bit precision. [17] uses efficient approximations of convolutional layers to generate binary-weight-networks. The weights representing the layers in the DNN are constrained to have

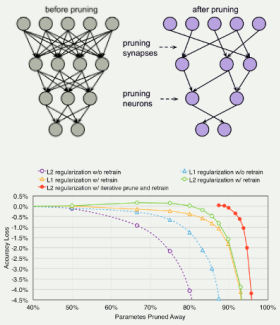
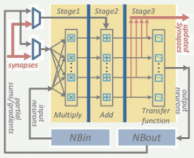
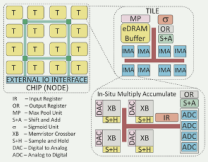
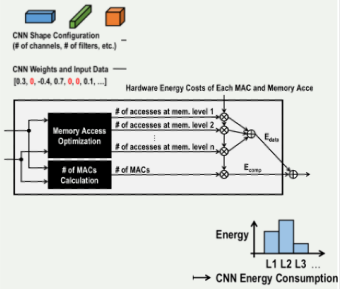
	Algorithmic Approach	Hardware Approach	Co-design
Solutions	• Model Compression • Smaller Networks	• Near memory architecture • In-memory architecture	• Low-bit networks • Energy aware training
Example	• **Pruning** Reduced synapses by 9× to 13× without loss of accuracy.  	• **DaDianNao** speedup of 450.65x over a GPU  • **ISAAC** – Memristor arrays Improvements of 14.8×, 5.5× in through-put and energy over DaDianNao 	• **Energy aware pruning** Energy consumption of AlexNet and GoogLeNet is reduced by 3.7× and 1.6×, respectively. 
Reference	Han, Song, et al NIPS 2015	Shafiee, Ali, et al. 2016	T.-J. Yang et al CVPR2019

Figure 1.3: Brief overview of the energy-efficient ML system

binary values reducing the memory requirements.

The second category is to directly train smaller or lightweight networks [20, 21, 22]. They assume no constraint on the underlying hardware architecture these algorithms are implemented on. In [21], large convolutional filters are replaced by multiple smaller ones and down-sampling is carried out late in the network so that the convolution layers have large activation maps. A reduction of nearly 50 times in the network weights was reported without compromising on the accuracy. MobileNets [22] are based on a streamlined architecture that make use of depth-wise separable convolutions resulting in lighter deep neural networks. The trade-off between the latency and accuracy of the network is controlled by two simple global hyper-parameters.

1.2.2 Hardware approach

The second approach to the problem is from the hardware standpoint, designing energy-efficient DNN accelerators. Current CPU/GPU architectures are highly generic designs aimed at solving a variety of computational problems. These are temporal architectures where a centralized processing unit controls the computation unit such as ALUs. These

ALUs can not communicate with each other and have to fetch the required data from the memory hierarchy every time a Multiply-Accumulate (MAC) operation is carried out. A few years ago researchers realized that ML-specific compute architectures are required to further optimize the mapping of ML algorithms onto hardware architecture improving throughput and accuracy. The fundamental building blocks of DNN are the fully connected layers and CONV layers which can be highly parallelized. Spatial architectures are proposed which take advantage of the high data reuse characteristics of these building blocks by forming a chain of these ALUs (called Processing Elements - PEs). These PEs pass the data among each other without the need for a control unit dictating every data movement. The required data is fetched from the memory hierarchy once and is passed onto these chains of PEs until all the processing related to the data is not completed. The routing of the data from the memory and within the PEs is called dataflow.

Based on these spatial architectures, DNN accelerators have been designed which varies in the selection of dataflow, signal nature (analog or digital) and the underlying device characteristics used as the building blocks for the PEs [23, 24, 25, 26, 27, 28]. The two broad categories are the near-memory and in-memory architectures. In near memory DNN accelerators [23, 28, 27], memory banks and compute units are separate entities. The architecture of [28] contains memory buffers for input/output neurons and weights and a Neural Functional Unit (NFU) which is largely a pipelined version of the typical digital PEs. The NFU consists of three stages. The first stage carries out the synapse multiplication, wherein the second stage binary tree adders are used to sum the result. The final stage consists of the application of a non-linear activation function. Eyeriss [27] is an accelerator for compact and sparse neural networks. A hierarchical mesh of PE nodes is used to address different types of DNN layers with varying data re-use and bandwidth requirements.

In-memory DNN accelerators are generally analog signal architectures where the memory used to store the synapses also acts as the compute unit. ISAAC [24] designs a complete pipelined DNN accelerator using memristor crossbar arrays performing dot product in an

analog manner. eDRAM is used to store the the output of current layer before it is fed into the crossbar arrays assigned to the next layer. Intra layer pipelining is used to improve the latency of the accelerator. RAPIDNN [29] carries out a transformation between the neuron to the memory with the aim of accelerating DNNs in a highly parallel architecture. The key point is the extraction of DNN weights and input values using a method of clustering in order to optimize the model for in-memory processing.

1.2.3 Algorithm-hardware co-design

The most interesting approach that recently has been surfaced is the co-design of ML system considering both algorithmic and hardware characteristics of the underlying system. Such Algorithm-hardware co-designs explore the device characteristics and hence design the high-level algorithm by taking into account the limitations posed by the hardware. [30] uses an approach of hardware-aware training, where once the underlying hardware architecture is defined, training is carried out incorporating the hardware constraints into the training procedure. This results in a DNN which is highly optimized (in terms of FLOPS and hence energy) for the selected hardware architecture. [31] introduces energy-aware pruning where based on the energy consumption, the next layer to be pruned is decided and has shown to perform better than their FLOPs-aware pruning counterpart. [32] uses Bayesian optimization to co-design deep neural network parameters and accelerator hardware parameters simultaneously maximizing the accuracy of the DNN and the energy efficiency of the hardware.

1.3 Unmanned Aerial Vehicle - An emerging IoT

Over the past decade, unmanned aerial vehicles (UAV) are emerging as a new form of IoT device being used in varied applications such as reconnaissance, surveying, rescuing, and mapping. Irrespective of the application, navigating autonomously is one of the key desirable features of UAVs both indoors and outdoors. Several solutions have been proposed

to make drones autonomous in an indoor environment. There has been significant work towards using additional dedicated sensing modalities such as RADAR [33] and LIDAR [34], which provide high accuracy in navigation and obstacle avoidance, thus enabling autonomous flights possible. But when the payload, cost, and power are taken into account, such systems are heavy, expensive, and power-hungry, making them almost impossible to be used in low-cost Micro Aerial Vehicles (MAV). Ultrasonic SONAR is a cheap alternative but suffers from a lack of accuracy and reduced field of view (FOV). They are also line of sight sensors that need to function in an array to provide a depth map. On the other hand, over the last decade, there has been significant interest in the use of Neural networks (NN) for various robotic applications. In recent years, reinforcement learning (RL) has been extensively explored for enabling a wide array of robotic tasks. The model-free nature of RL makes it suitable in problems where little or nothing is known about the environment. RL has been successfully implemented in games and has shown beyond human-level performance [35]. However, RL is a data-hungry method and often requires more data compared to other machine learning techniques to generate comparable results.

In the case of RL for real-time collision avoidance, a major latency bottleneck arises from the need to train a CNN with the current image frame, which must be completed before the next image frame is captured. This is illustrated in Figure 1.4 where we show the relationship between the speed of a drone and the required frame per second (fps) of the image acquisition system. As shown in Figure 1.4(a), for a given velocity of the drone, we can calculate the minimum fps requirement of the camera for collision avoidance based on the corresponding distance traveled between two frames (d_{frame}), and the minimum distance between the drone and its obstacles (a measure of clutter in the environment). From Figure 1.4(b), we observe that the fps requirement increases as the speed of the drone increases. Since the minimum distance between a drone and its obstacles is lower in typical indoor environments compared to outdoor environments (i.e., the indoor environment is more cluttered than outdoor environments), drones in indoor environments require higher

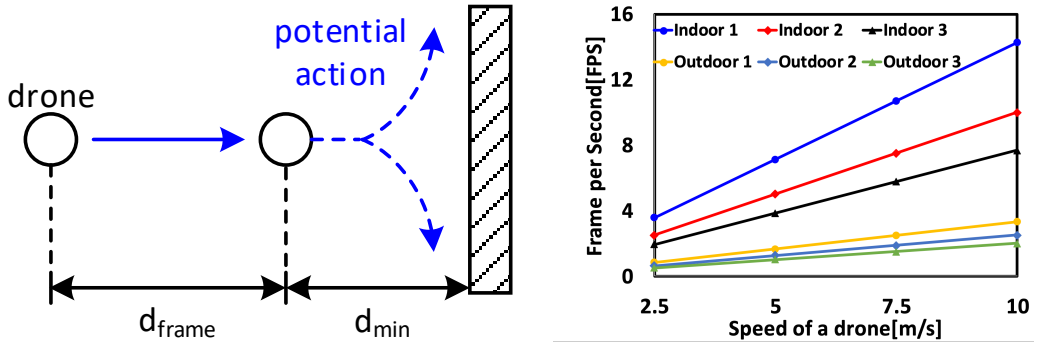


Figure 1.4: Relationship between the speed of a drone and required fps

fps compared to outdoor environments. As the fps increases, the time available to perform real-time RL decreases necessitating the high-performance of the computing system. For small power-constrained drones, it requires significant hardware resources to execute the training process in RL within the latency and power targets

The performance of machine learning algorithms depends heavily upon the complexity of the network and the amount of meaningful data available for training. For a complex task, in general, the deeper the NN, the better the performance (until we hit a limit). Correspondingly, the amount of meaningful data scales too [36] until the point where the task is not complex enough given the network architecture and performance starts degrading [37]. Training a deeper neural network comes with the cost of increased computations. This makes it challenging to be implemented on a limited-resource edge node such as a mobile drone. Simpler NNs with real-time training can be implemented on edge nodes, but this is achieved only by compromising the performance of the underlying application. So, for acceptable performance, the network should be deep enough, which comes with:

- Additional compute requirement
- Increased Power consumption
- Increased latency

For a resource-constrained edge node (like a lightweight drone), an additional compute resource means adding more hardware to the drone decreasing its thrust-to-weight ratio.

An increased amount of power consumption may drain the battery quicker rendering the drone useless and increased latency will affect its response time making it far from being real-time. Hence these additional requirements are in direct contrast with the drone's inherent limitations. Simpler NNs require a reduced amount of computations and are possible to be implemented on edge nodes. But for a complex enough task, these simpler NNs do not perform well. So the problem is, for RL-related applications how can we implement neural network training on resource-constrained edge nodes without losing too much performance and with reduced power and latency. One direct approach is to use Offline Training and Deployment i.e. training the NN on the cloud, and carrying out inference on the edge nodes. For tasks involving supervised learning (say classification), this is an effective solution. But for RL-related problems, where there is no clear boundary between the training and inference phase, this can't be implemented directly. [38] however, uses an approach where the network is trained on simulated environments posing RL as a supervised learning problem and then deployed on new unknown environments. This transfer of knowledge without further fine-tuning doesn't always work well and is closely tied with the co-relation or similarity between the train and test environments. The more the similarity between the training and testing environment the better the performance and vice-versa. [39] learns a CNN with regressors using supervised learning to follow a pre-determined path and fails to perform if the environment changes.

1.4 Background on Deep Reinforcement Learning

RL is one of the three basic machine learning paradigms (the other two being supervised and unsupervised learning) where an agent interacts with the environment taking actions to maximize the notion of cumulative rewards. The key objective is to learn a control policy, i.e. optimal state-space to action space mapping, that when implemented results in maximizing the underlying goal of the agent. As opposed to Supervised Learning (SL) where the target labels are known, in RL we do not have access to labeled data points.

Rather, the agent explores the environment by taking actions (random at first) receiving rewards that can be used to quantify the nature of the action taken to be a good or a bad action. RL can be modeled as a Markov Decision Process (MDP) where the next state of the system only depends on the current state and the current action taken. The RL MDP can be defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}$ is the MDP transition probabilities, $\mathcal{R}$ is the reward function and γ is the discount factor (details below).

The agent interacts with the environment $\mathcal{E}$ in a sequence of actions $\{a_0, a_1, \ldots\}$, observations $\{s_0, s_1, \ldots, \}$, and rewards $\{r_0, r_1, \ldots\}$. At each time instant t, the agent observes the current state $s_t \in \mathcal{S}$ where $\mathcal{S}$ is the state space of the MDP i.e. a set of all the possible states. Based on the current state s_t, the agent takes an action $a_t \in \mathcal{A}$ from a pre-defined set of actions and a reward $r_t \in \mathcal{R}$ is observed. The reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a mapping from the current state s_t and action a_t to a scalar r_t. The reward function $\mathcal{R}$ is designed to quantify the underlying goal and is a design parameter. The action a_t is then implemented in the environment and based on the the transition probabilities $\mathcal{P}$ a new state s_{t+1} is observed. The state transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the probability that the agent will move from state s to another state $\hat{s}$ under the action a i.e. $\Pr[s_{t+1} = \hat{s} | s_t = s, a_t = a]$. At any iteration t, the objective of RL is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$, i.e. a mapping from the current state to action, to maximize the discounted return starting from the state s_t. The discounted return is given by

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1.1}$$

where $\gamma \in (0, 1)$ is the discount factor that controls the importance between current and future rewards. Given a policy π, each state can be assigned a state value

$$V^{\pi}(s) = \mathbb{E}(R_t | s_t = s) \tag{1.2}$$

which is defined as the expected return starting from the state s and following the policy π. Similarly, each state-action pair can be assigned an action value under the policy π given by

$$Q^{\pi}(s,a) = \mathbb{E}(R_t|s_t = s, a_t = a) \tag{1.3}$$

In this document, we will consider the model-free RL, where the transition probability function $\mathcal{P}$ is unknown but can be observed through sampling. Both the value-based and policy-based methods will be considered to solve the RL problem.

In value-based methods, the RL problem is solved by finding a policy that maximizes the value function. Bellman optimality equation can be used to update the action values Q for sampled trajectories/sequence which is given by

$$Q(s,a) = r + \gamma max_{a'} Q(s',a') \tag{1.4}$$

With the repeated application of the Bellman optimality equation, the action values Q converges. The policy π is then given by

$$\pi(s) = \arg\max_{a} Q(s,a) \tag{1.5}$$

On the other hand, instead of maximizing the value function and then defining the policy in terms of the value function, policy-based methods directly try to infer the optimal policy without explicitly calculating the value function. The policy is modeled with the parameter θ i.e. $\pi_{\theta}(s)$. The problem then becomes in finding the optimum θ that leads to an optimum policy maximizing the value function i.e.

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{s\sim\rho} V^{\pi_\theta}(s). \tag{1.6}$$

where ρ is the initial state distribution over the action space. In the rest of the document, we will use both value-based and policy-based methods. Explicit details will be provided whenever any of these methods are used.

1.5 Dissertation Overview

The rest of the document is organized as follows. An introduction and formal definition of the application of drone autonomous navigation using reinforcement learning is introduced. This problem will be used as the underlying application to test the proposed energy-efficient methods. It is shown that conventional vanilla algorithms can not be directly implemented on compute-limited drones and need to be modified to take into account their limitations. Then we will move on to proposing different methodologies to address the issue of energy efficiency by using both an algorithmic approach (using transfer learning) and by designing an energy-efficient in-memory ML accelerator using novel STT-MRAM. The scope of the RL application is then increased from single-agent to multiple agents in a distributed setting and the issue of adversaries in such a multi-task RL problem is addressed. Finally, an open-source benchmarking tool for drone RL applications is discussed that was developed through this research, before concluding the document.

CHAPTER 2

DRONE AUTONOMOUS NAVIGATION IN REAL ENVIRONMENTS USING RL

In this chapter, we will define the problem of drone autonomous navigation as an RL problem. We will look at the data, safety, and energy challenges of implementing RL in the context of drone navigation in real environments and address the issues by proposing a few solutions.

2.1 Introduction

We explore a single-camera-based autonomous navigation and obstacle avoidance for MAVs in real environments. Traditional systems employ handcrafted sensing and control algorithms to allow navigation and have led to significant progress in this field [40, 41]. Recently, the success of deep neural networks has enticed researchers to study neuromorphic models of autonomous navigation [42, 43, 44]. In spite of the success of such machine learning models, we also recognize that true autonomy in intelligent agents will only emerge when bio-mimetic systems can perform continuous learning through interactions with the environment.

The main contributions are as follows:

- Demonstration of end-to-end Deep RL for collision avoidance using monocular images only and without the use of any other sensing modality.
- Overcoming the issues associated with the implementation of RL in real environments by designing a suitable reward function that takes into account both the safety and sensor constraints.
- Using expert data and knowledge-based data aggregation to improve the RL convergence in real-time.

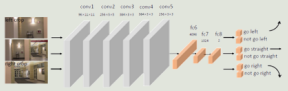
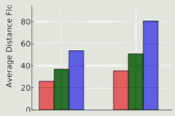
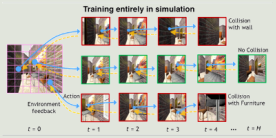
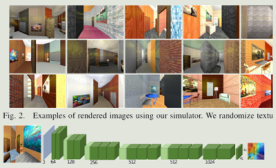
	Learning to fly by crashing	Monocular UAV control	CAD²RL
Detail	• Supervised Learning • Data driven approach to learn to fly by crashing more than 11,500 times in a multitude of diverse training environments.  	• RL mechanism to navigate a drone in a forest • Imitation learning DAgger Algorithm • Preprocessing using handcrafted features before it is fed to the network  	• Reinforcement Learning • Training in simulated 3D environments • VGGNet as deep neural network  
Cons	• Not model free	• Handcrafted features	• Challenges in real world
Reference	Gandhi et al IROS 2017	Ross et al ICRA 2013	Sadeghi et al 2016

Figure 2.1: Brief overview of ML based drone autonomous navigation

2.2 Related Work

Our principal goal is to enable the UAV to fly by itself in a real environment, without incurring any additional hardware or sensor cost. Most of the low-cost MAVs come equipped with an onboard camera and an Inertial Measuring Unit (IMU). So the use of image frames for navigation is an area of active research. We have studied supervised learning for drone navigation. [45] collects a data-set of 11,500 videos of crashing and learns a neural network that classifies an image as “crash” or “no crash”. Based on that knowledge, the authors use a handcrafted algorithm to steer and navigate the drone away from obstacles. [46] uses an indoor data-set and classifies the images according to the action taken by the drone. They define a set of five actions in the action space of the drone, hence posing the problem as a classification problem with five classes. A supervised image classifier with three classes is used [47] in to train a deep neural network for forest navigation. The data set is collected by mounting three cameras on a hiker’s head facing forward, left, and right. [48] uses RL as the online learning mechanism to navigate a drone in a forest. A camera frame is taken and is pre-processed before it is fed to the RL system. This pre-processing uses handcrafted algorithms to extract lower dimensional features from the camera image. [38] uses simu-

lated environments with a larger set of action space (1681 actions). The agent is trained for a deep neural network in 9 simulated environments and the performance is reported. The neural network trained in the simulations is then also tested in the real environment without any fine-tuning and has shown to perform well. Unfortunately, the performance of this approach greatly depends upon the correlation of the simulated and real environment. For the cases of unknown environments which has limited similarity with the simulated training environments, the agent is expected to behave poorly.

All of these previous demonstrations and approaches, in spite of their many successes, either require considerable human/expert intervention, handcrafted algorithms or are implemented offline in simulations, where the simulated and the real endowments need to be nearly identical.

2.3 Problem formulation

The objective of drone autonomous navigation via RL is to learn a control policy by interacting with the environment. The idea is to take actions that lead to a collision-free flight of the drone in a real environment. There is no goal position and the objective is to navigate through the arena safely. Consider the task of obstacle avoidance where the drone interacts with the environment in a sequence of actions, observations, and reward calculations. At each time instant, the drone observes the current camera frame s. It takes an action a from an action space $\mathcal{A}$ and implements it. Implementing the action moves the drone to a new position where it observes a new camera frame s'. This new camera frame along with the action taken will quantify a reward r. The goal of the system is to take actions maximizing the long term reward, i.e. at each time step t, we need to take an action that eventually leads us to a sequence of states s_i with rewards r_i for $i \in \{t+1, t+2, ...\}$ such that the future discounted return $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ is maximized, where $\gamma \in (0, 1)$ is the discount factor. Each of the state-action pair is assigned a Q value $Q(s, a)$. During the learning phase these

Q values are updated according to the Bellman optimality equation as follows

$$Q(s, a) = r + \gamma max_{a'} Q(s', a') \tag{2.1}$$

Bellman equation update ensures that in a given state s_t selecting an action $a_t = max_{a'} Q(s_t, a')$ will result in maximizing the future discounted reward R_t. These Q values are stored as an approximation of a function with states as input. In Deep Reinforcement Learning (DRL) the function to estimate these Q values is a deep neural network.

2.4 Challenges of implementing DRL in real environments

RL in real environments for collision avoidance is challenging. The methodologies adopted in this paper to address them are described in the next section.

2.4.1 Reward generation

In real environments, the position of the agent and its distance from obstacles is not known. Hence extra sensing capabilities need to be added to the agent giving it a notion of depth which not only adds to the computation cost but also to the weight of the agent. In this paper, we demonstrate DRL using a single monocular camera.

2.4.2 Safety issues

RL works via a trial and error method. It is designed to learn from mistakes. For the task of collision avoidance, it means that the agent has to collide with the obstacles to learn. This collision can not only harm the agent, but also the environment. We propose a method of *virtual crash* and a *crash reward* to address this issue.

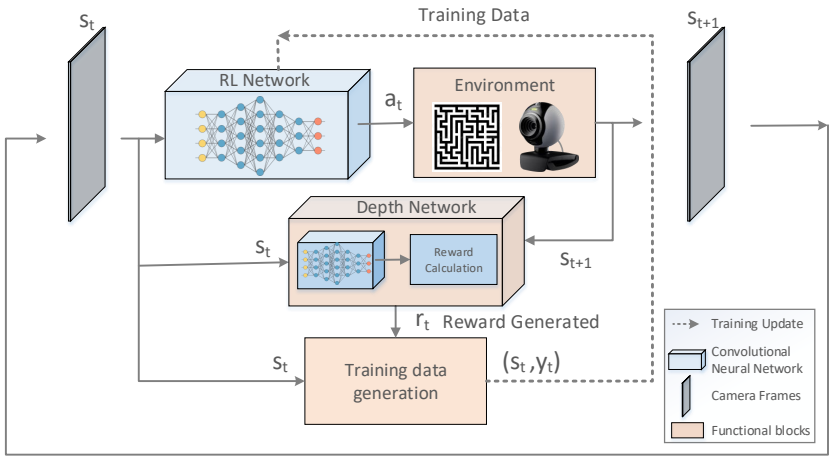


Figure 2.2: Block diagram of the key algorithmic components that enable end-to-end RL for obstacle avoidance and autonomous flight in a drone.

2.4.3 Resetting the agent to a suitable initial position

RL requires that the agent must be placed at the proper initial position (usually the same) every time it crashes with an obstacle. In simulations, it is trivially achieved while in real environments it poses a challenge. We demonstrate a method of *un-doing the drone's actions* to achieve the same effect as resetting the drone's position.

2.4.4 Large online data-set requirement

The amount of data required for implementing RL is large. Such training data requirement stems from the fact that the agent starts with little knowledge of the environment and takes random actions to explore it. As opposed to simulations where you can easily collect a large number of data points, the data-set that can be collected in a real environment is limited. We use several techniques to address this issue, as described in the next section.

2.5 Navigation in Real Environments via RL (NAVREN-RL)

We propose an end-to-end drone navigation methodology using expert data-aided deep reinforcement learning on images acquired by a single camera. The end-to-end approach has been summarized in the block diagram shown in Figure 2.2. We limit the action space

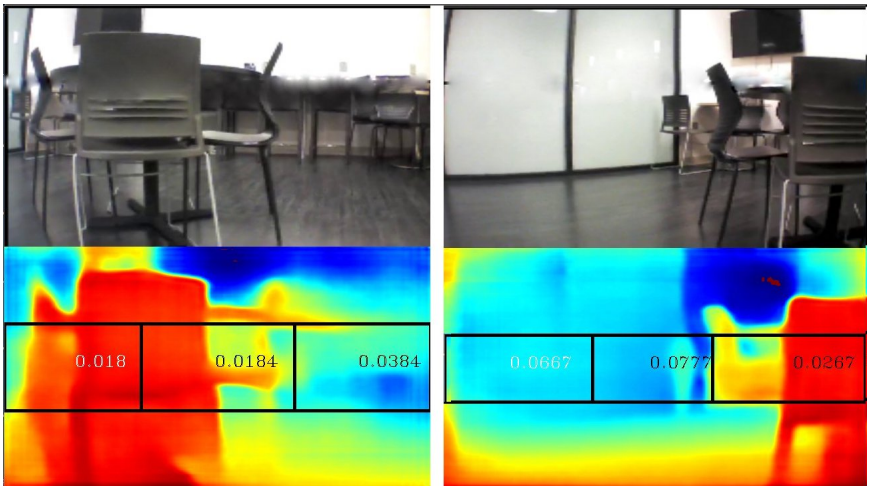


Figure 2.3: Depth-based dynamic windowing

to three actions $\mathcal{A} = \{a_F, a_L, a_R\}$ where under the action a_F the drone moves forward (by $0.25m$), a_L the drone turns left (45^o) and a_R the drone turns right (45^o). To address the issues of real-time DRL, we explore the following solutions keeping in mind the agent's weight, cost, limited sensing capabilities, and environmental constraints.

2.5.1 Reward generation

Since we are not using any external sensing modalities, the reward needs to be generated from the image frame itself. We use the depth map of the state towards the generation of the reward. A depth map of a frame is an image of the same dimension with pixels intensities corresponding to the depth of the pixel in the input image. In the last few years, various offline learning algorithms have been explored to generate depth maps using a single image [49, 50, 51]. Due to its superior test accuracy, we use the approach proposed in [50].

In order for the reward to be simple and meaningful, we use parts of the depth map towards reward generation. The depth map generated is divided into three windows. The depth in the windows is used to generate a notion of the distance to the closest obstacle in each of the three (left, center, and right) directions. This distance is calculated by averaging the smallest $n\%$ pixel depth values. The value of n depends on the nature of the

Algorithm 1: Reward generation using the depth map

Function `RewardFunction(`s_t, a_t, s'_t`)`**:**

$d(s_t) \leftarrow$ depth map of s_t
$d(s'_t) \leftarrow$ depth map of s'_t
$d^l(s_t), d^c(s_t), d^r(s_t) = DepthValues(d(s_t))$
$d^l(s'_t), d^c(s'_t), d^r(s'_t) = DepthValues(d(s'_t))$

if $a_t = a_F$ **then**
  $r_t = d^c(s'_t)$
else if $a_t = a_L$ **then**
  $r_t = d^c(s'_t) + \alpha(d^l(s_t) - d^r(s_t))$
else
  $r_t = d^c(s'_t) + \alpha(d^r(s_t) - d^l(s_t))$

if $d^c(s'_t) < d_{thresh}$ **then**
  $r_t = r_{crash}$
return r_t

environment. If the environment is expected to have narrow (wide) obstacles, the value of n is relatively smaller (larger). We note that changing the window size dynamically with the global depth in the scene aids reward generation and improves accuracy. If the global depth of the image is greater, then the objects being seen in the frame are farther apart. We choose the relationship between the global depth and window size empirically to be $[H,\ W]/(0.75 \times \textit{global\_depth} + 0.5)$ where $[H, W]$ are the dimensions of the input frame from the camera. This global depth-based dynamic windowing can be seen in Figure 2.3. The three local distances to the closest obstacle in corresponding directions are then put to use towards reward generation according to algorithm 1 where $\alpha \in [0, 1]$ is a parametric weight and is taken to be 1/3; d_{thresh} is used to mark the completion of an episode as explained in the next section.

2.5.2 Addressing safety issues

If at any point, the center window shows the distance to the nearest obstacle d^c to be below some threshold value d_{thresh}, the agent stops and considers to have "virtually crashed". This virtual crashing marks the end of an episode. Thus the agent does not physically collide

with obstacles and significantly reduces the risk of damaging itself or the environment. Once the agent virtually crashes, a penalizing reward r_{crash} is provided to the state-action pair leading to the crash.

2.5.3 Resetting the agent to a suitable initial position

In our approach, the agent does not reset to its initial position, rather a new initial position is selected after the end of every episode. The new initial position is chosen in an autonomous way making use of the knowledge of the "virtual crash" state-action pair. The action that led to the collision is undone. The agent accomplishes this by taking the opposite actions (for e.g. if the forward action led to the virtual crash, the agent after marking it the end of an episode, moves backward) until d^c is at least $d_{recover}$; a threshold set for recovering from the crash. A new episode starts from the recovered state and the policy prevents the agent from selecting the "virtual crash" action for that initial state.

2.5.4 Large online data

Expert Data $D_\mathcal{E}$: We address the requirement of a large training data set by making the use of Learning from Demonstration (LfD) [37]. At the onset, a human expert navigates the agent across the arena and collects a limited set of expert data points. The idea of collecting expert data points is to help the agent through guided exploration. This expert data set is used towards learning in the following two ways.

- Pre-training phase: The neural network is trained for this small set of expert data $D_\mathcal{E}$ and the weights learned $\theta_\mathcal{E}$ are used as initial weights for the online learning phase. This preserves some knowledge about the environment and gives the agent a good starting point for exploration.

- Expert data as a part of experience replay: The expert data is also used as a part of the replay memory $\mathcal{D}_{replay}$ from which the batches of data points are sampled for

training. Making expert data a persistent part of the experience replay helps avoid the neural network from forgetting what it had learned in the pre-training phase.

Knowledge-based Data aggregation: The data aggregation is carried out in the following two ways:

- When the agent virtually crashes, going forward from that state will lead to a crash too. If the agent which is in state s_t moves to the next state s'_t by taking an action a_t and virtually crashes, then the data-point $(s'_t, a_F, s'_t, r_{crash})$ will be aggregated to the current data points.

- Since opposite actions are selected to recover from crashes, the intermediate states will lead to a crash as well. For example, the agent in state s_t moves to next state s_{t+1} by taking an action a_t and virtually crashes. Let a'_t be the opposite action to a_t. If $a_t \in \{a_R, a_L\}$ then the data-points $(s_{t+i}, a'_t, s_{t+i-1}, r_{crash})$ and $(s_{t+i}, a_F, s_{t+i}, r_{crash})$ for $i = \{1, 2, 3, ..., n_{recover}\}$ is aggregated to current data-points where $n_{recover}$ is the number of steps required to recover from the crash. Since going backward does not belong to our defined action space $\mathcal{A}$, the data points are not aggregated if $a_t = a_F$

2.5.5 Convergence of Deep RL algorithm

The basic RL algorithm often suffers from limited convergence, which mainly emerges because the Bellman equation tends to over-estimate the Q-values due to its non-linear nature. Also, the aggregating nature of the Bellman equation might lead to diverging Q-values. So, in order to avoid these issues, the following solutions are implemented.

Restricting the range of rewards

The distances to the nearest obstacle $\{d^l, d^c, d^r\} \in \mathbb{R}^+$ is the estimated distances in meters. These distances are scaled down to have values between $[0, \frac{1}{\alpha+1}]$ where α is the weight

constant used in the reward function. When scaled down, the reward function generates the reward within the limited range of $[-1, 1]$

Clipping Q values in Bellman equation

This ensures that the Q-value updates do not diverge. Let $Q_\theta^{target}(s, a) = r+\gamma \max_a Q(s', a; \theta)$ be the normal Q-value update where θ is the weights of the neural network, then the clipped Bellman equation is

$$\hat{Q}_\theta^{target}(s, a) = clip(Q_\theta^{target}(s, a), -1, 1) \tag{2.2}$$

where the function $clip(a, n_1, n_2)$ clips the value to n_1 or n_2 if a is less than n_1 or greater than n_2 respectively. The updated equation ensures that $Q_\theta^{target}(s, a) \in [-1, 1]$ and does not diverge.

Use of Double DQN

We address the overestimation of the Q value by using a Double Deep Q Network (DDQN) [52]. In DDQN two different copies of the neural networks are used. One of the neural networks (the behavior network, θ) is used for training, while the other network (the target network,θ') is used towards the Bellman equation update. The target network is updated with the weights of the behavior network after every n_{target} interval. The updated Bellman equation looks like

$$Q_{\theta'}^{target}(s, a) = r + \gamma \max_{a'} Q(s', a'; \theta') \tag{2.3}$$

Combining both clipping and DDQN, the updated Bellman equation is:

$$\hat{Q}_{\theta'}^{target}(s, a) = clip(r + \gamma max_a Q(s', a; \theta'), -1, 1) \tag{2.4}$$

Algorithm 2: NAVREN-RL Algorithm

Input: Expert data-points: $D_{\mathcal{E}}$
Initialization: Behaviour network: $Q_\theta(s) = \mathcal{N}(s;\theta)$, Target network: $Q_{\theta'}(s) = \mathcal{N}(s;\theta')$, m: Number of pre-training updates, n_{target}: Target network update interval, b_ϵ:ϵ annealing coefficient, n_{batch}: mini-batch size for training
for $i \in \{1, 2, 3, ..., m\}$ **do**
    Sample a mini-batch of size n_{batch} from $D_{\mathcal{E}}$
    Evaluate the loss $J_{\theta'}(\theta)$
    Perform gradient descent to minimize $J_{\theta'}(\theta)$ w.r.t θ
end
Initialize the replay memory $\mathcal{D}_{replay} \leftarrow D_{\mathcal{E}}$
for $t \in \{1, 2, 3, ...\}$ **do**
    $s_t \leftarrow$ *Camera image*, $Q(s_t) \leftarrow \mathcal{N}(s_t;\theta)$
    Sample an action from behaviour policy $a_t \sim \pi^{b_\epsilon Q}(\epsilon)$
    Implement the action a_t on the agent
    $s'_t \leftarrow$ *Camera image*, $Q(s'_t) \leftarrow \mathcal{N}(s'_t;\theta)$
    Generate the reward $r_t \leftarrow \texttt{RewardFunction}(s_t, a_t, s'_t)$
    Store the tuple (s_t, a_t, s'_t, r_t) in $\mathcal{D}_{replay}$
    if *virtual crash* **then**
        while *not recover from crash* **do**
            Aggregate data-points to $\mathcal{D}_{replay}$
        end
    end
    Sample a mini-batch of size n_{batch} from $\mathcal{D}_{replay}$
    Evaluate the loss $J_{\theta'}(\theta)$
    Perform gradient descent to minimize $J_{\theta'}(\theta)$ w.r.t. θ
    if t *mod* $n_{target} = 0$ **then**
        $\theta \leftarrow \theta'$
    end
end

2.5.6 Network Architecture

We use a modified AlexNet [53] network to estimate the Q values for the states. The input to the network is the re-sized camera frame s_t. The network consists of 5 convolutional layers and 3 fully connected layers.

2.5.7 Online Learning

Before the learning process begins, an expert user navigates the agent in the selected environment for a certain number of steps n_{expert}. The data tuple (s_i, a_i, s'_i, r_i) for each of the steps $i \in \{1, 2, 3, ..., n_{expert}\}$ is generated and saved in $D_{\mathcal{E}}$. Next comes the pre-training phase where random mini-batches of size n_{batch} are selected from the expert data $D_{\mathcal{E}}$ and a neural network $Q_\theta(s) = \mathcal{N}(s; \theta)$ is trained minimizing

$$J_{\theta'}(\theta) = \sum_{i=1}^{n_{batch}} J(s_i, a_i, \theta, \theta') + \beta J_{reg}(\theta) \tag{2.5}$$

where $J(s_i, a_i, \theta, \theta')$ is the TD loss for i^{th} data-point dictated by the Bellman equation and $J_{reg}(\theta)$ is regularization loss to help prevent over-fitting the network for the smaller amount of expert data, and β is a regularization weight. These losses are given by:

$$J(s_i, a_i, \theta, \theta') = ||\hat{Q}^{target}_{\theta'}(s_i, a_i) - Q(s_i, a_i; \theta)||_2 \tag{2.6}$$

$$J_{reg}(\theta) = ||\theta||_2 \tag{2.7}$$

where $\hat{Q}^{target}_{\theta'}(s_t, a_t)$ is given by Equation 2.4

After the pre-training phase, the online training phase begins. The agent is placed in the environment and follows a ϵ-greedy policy for actions. with b_ϵ as the annealing coefficient. ϵ is varied linearly from 0.1 to 0.9 as the number of data points varies from 1 to b_ϵ. At every time step t, the drone saves the data points (s_t, a_t, s'_t, r_t) in $\mathcal{D}_{replay}$. A mini-batch of size n_{batch} is randomly sampled from the replay memory $\mathcal{D}_{replay}$ and used to minimize the loss defined in Equation 2.5 through gradient descent. algorithm 2 shows the complete algorithm, while Table 2.1 lists the hyperparameters used.

Table 2.1: List of hyper parameters used for training

Learning rate	1e-6	n_{target}	200	n_{batch}	32
β	0.001	d_{thresh}	0.02	r_{crash}	-1

2.6 Experimental Results

Real-time experimentation is carried out to validate the proposed approach for drone navigation.

2.6.1 Hardware specifications

We use a low-cost Parrot AR drone 2.0 which does not have the computational power to carry out the required processing onboard. Hence, the drone sends the camera frames to a workstation/cloud equipped with a Core i7 processor and GTX1080 GPUs. Control actions are communicated back to the drone. We use Tensorflow to carry out the neural network computation on the workstation.

2.6.2 Testing environments

We use the following two arenas to carry out the experimentation for successful navigation.

Arena A_1: Open Hallway

This is a hallway in an engineering building with glass walls. The drone has to navigate through the narrow hallways (minimum width of $\approx 1.5m$). There are no extra obstacles between the hallway path except for the water dispenser, benches, and trashcans.

Arena A_2: SC Room

This arena is a cluttered break-out room with couches, chairs, tables, and bar-stools with narrow passages in between ($\approx 1m$).

The layout and floor plans of these arenas can be seen in the Figure 2.4

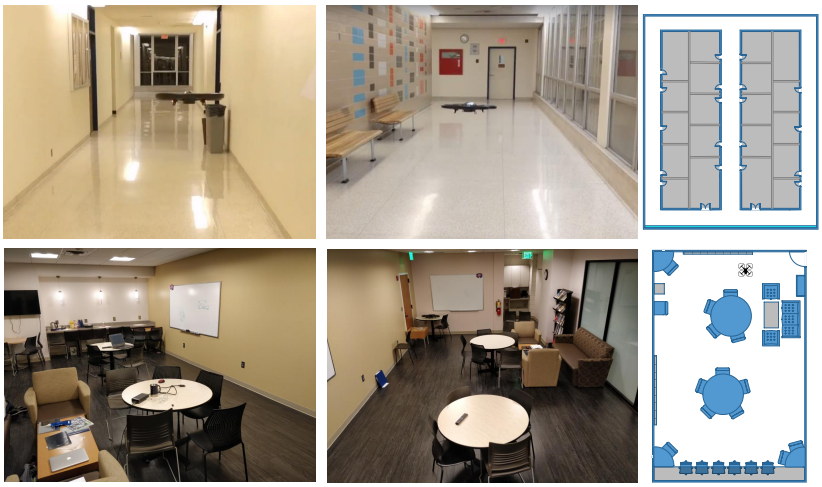


Figure 2.4: Snapshots and the layouts of the arenas used. Top row: A_1 Hallway, Bottom row: A_2 SC-room

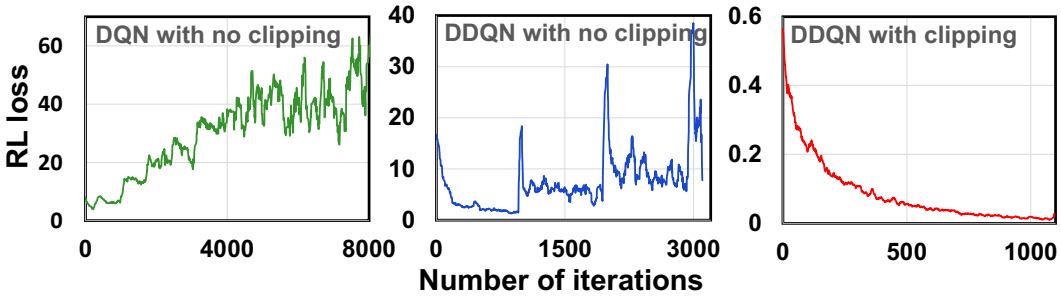


Figure 2.5: Convergence of RL with and without DDQN and clipping

2.6.3 Baseline Algorithms for Comparison

We compare our method with the following baseline algorithms.

Straight-line controller

This controller always predicts the forward action hence moving the agent in a straight line in a manner described in [38]. This controller provides a good comparison of the complexity of the arena.

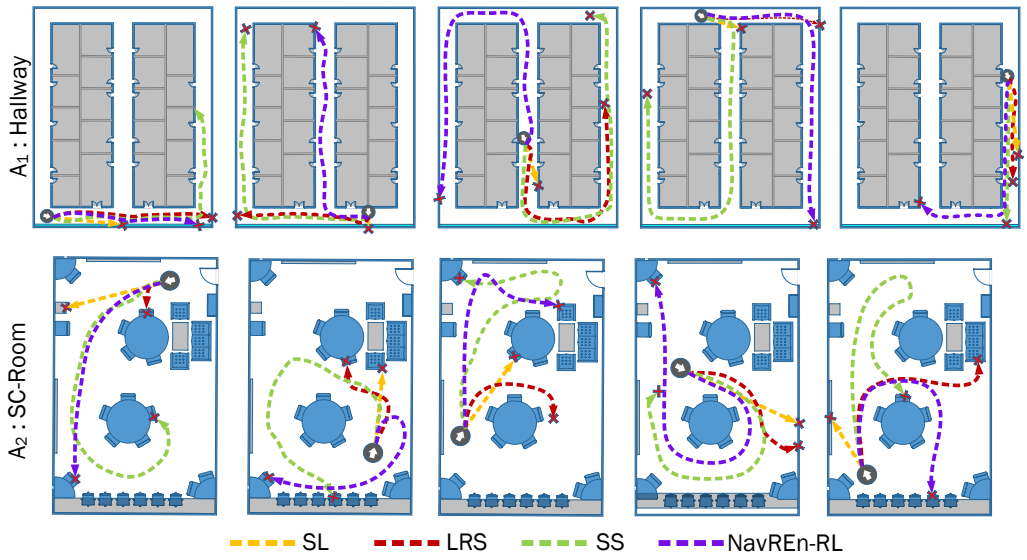


Figure 2.6: Trajectories followed by the baselines and NAVREN-RL for 5 different initial locations

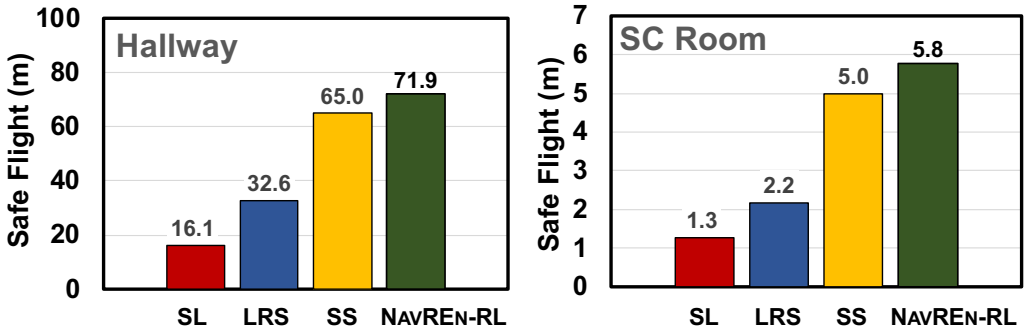


Figure 2.7: Safe flight (in meters) comparison across baselines

Left-Right-Straight (LRS) controller

This baseline is based on the work in [48] where a supervised approach is used to classify images with respect to the actions required to be taken. A human expert roams around the arena and collects the images using left, right, and forward-facing cameras. Images collected from the left (right) facing camera are labeled with the target action of right (left) while the ones collected from the forward-facing camera are labeled with the target action of forward. These labeled images are then used to train a neural network offline in a supervised manner.

Table 2.2: Arena stats

Arena	Method	Total Distance(m)	Safe Flight(m)	Improvement
Hallway	**SL**	80.7	16.1	4.45x
	LRS [48]	162.9	32.6	2.21x
	SS [45]	324.9	65.0	1.11x
	NAVREN-RL (ours)	**359.5**	**71.9**	–
SC-room	**SL**	6.3	1.3	4.55x
	LRS [48]	10.9	2.2	2.65x
	SS [45]	24.9	5.0	1.16 x
	NAVREN-RL (ours)	**28.8**	**5.8**	–

Self-supervised (SS) controller

This controller uses the work proposed in [45] where a large data-set of the crash and safe images are collected over various indoor environments. These labeled images are then used to train a neural network to classify each image as either safe or crash. In the inference phase, the input camera frame is then divided into three windows and the probability of a crash in each of the sub-frames is calculated. Based on these probabilities, a handcrafted controller is designed, following [45] to take suitable actions.

2.6.4 Performance

Figure 2.5 shows the comparison of RL convergence with and without DDQN and clipping of the Bellman equation. It can be seen that the DDQN with clipping of the Bellman equation shows good convergence.

We assess the performance of NAVREN-RL by comparing it against the baselines mentioned above. 3000 (700 expert+2300 online) data-points are collected in the Hallway arena, while 2000 (600 expert+1400 online) data-points are collected in the SC-room arena. In each of the arenas, all 4 techniques are separately used to learn a neural network. The agent is initialized by the learned neural network and the performance is evaluated by placing the drone at 5 different initial locations. To keep the comparison fair, the agent is placed precisely the same way across all the techniques. In each of the cases, the trajectory fol-

lowed by the agent is recorded until the agent is no longer able to navigate. This loss of navigation is considered if

- The agent collides into an obstacle
- The agent keeps hovering, being stuck in a repetitive pattern of left/right actions, and does not move forward for 10 iterations

The trajectories can be seen in Figure 2.6. The distance covered by the agent before the crash is taken to be the performance metric and can be seen in the Table 2.2. The total distance covered is the sum of the individual distances covered by the drone from each of the initial locations. The safe flight for any technique is the average distance covered by the drone from the different initial locations. In most of the cases, the proposed NAVREN-RL method outperforms the baselines, i.e the safe flight (m) for the proposed RL method is the highest among the baselines. This can be seen in Figure 2.7.

2.7 Summary

An end-to-end reinforcement learning algorithm for autonomous navigation of drones in indoor real environments by addressing the problems associated with the RL implementation was addressed. Experimentation was carried out in different arenas and the performance is compared to other base lines. The results show that the agent is able to navigate in the indoor arena with limited sensing capabilities and data points with comparable performance.

CHAPTER 3

AUTONOMOUS NAVIGATION USING TRANSFER LEARNING

Implementing drone autonomous navigation in real environments using the approach mentioned in the previous chapter results in a good enough performance. The issue however is the amount of energy required. Training the entire network on the fly makes it very energy and time-consuming. Hence we need to look at ways to address this. In the next chapter, we try to directly address this issue by making use of data transfer from simulation to the real world. This data transfer is supposed to be good enough for global decision-making, while the network still needs to be fine-tuned for local task-specific features.

3.1 Introduction

Since the overall objective is to make Micro Aerial vehicles (MAV) capable enough of carrying out ML training algorithms, this problem can be approached in either of the two areas. The first and more direct approach is to make better hardware engines for DNN accelerators [54, 55]. Authors of [56] design and implement an energy-efficient accelerator for visual-inertial odometry (VIO) that enables autonomous navigation of miniaturized robots. [57] demonstrates a navigation engine for autonomous nano-drones which is capable of closed-loop end-to-end DNN-based visual navigation. The other approach is to devise better and improved algorithms that take a lesser amount of computations (hence energy) for similar performance such as model compression [58, 11]. [12] developed Network Pruning, which begins with a pre-trained model, then the network parameters which are below a certain threshold are replaced with zeros forming a sparse matrix, and finally performs a few iterations of training on the sparse CNN. The downside of this approach is that the network needs to be iteratively pruned and re-trained until the desired compression is achieved. Moreover, this approach might not be useful for online ML problems such as

RL where re-training the network is not energy efficient at all. [21] presents SqueezeNet, a CNN architecture that has 50x fewer parameters than AlexNet and maintains AlexNet-level accuracy on ImageNet by exploring the design space of the convolutional network. This tiny network might be problem-specific and is not guaranteed to be complex enough for convoluted tasks such as end-to-end autonomous navigation. This chapter proposes an approach that falls in the latter category.

Transfer learning is a well-established approach to transferring any prior domain knowledge to a new problem or domain. This is how the human brain works, instead of learning any new problem from scratch, it uses pre-existing knowledge about prior problems and uses that along with learning a new skill set to solve the problem. Transfer learning has been widely used in Machine Learning problems to address the issues of smaller or insufficient amounts of data, mitigating convergence issues, reducing the time/steps required for convergence [59, 60, 61, 62, 63, 64, 65, 66]. These issues are addressed by learning a neural network for one task and using the learned weights as initialization to another network for a different task. The network weights are then fine-tuned based on the new domain knowledge (dataset). The most common and simplest example of TL is using Imagenet learned weights as an initializer for classification problems.

To the best of our knowledge, all the TL papers in the past discuss TL as a tool/approach to address the above-mentioned issues without worrying much about the computational cost required to train a deep neural network. In this chapter, we show we can use Transfer learning, to segment a deep network into trainable and non-trainable parts reducing the training computations, for the underlying tasks without compromising too much on their performance. This reduction in computation directly translates to reduced training energy leading to an energy-efficient system.

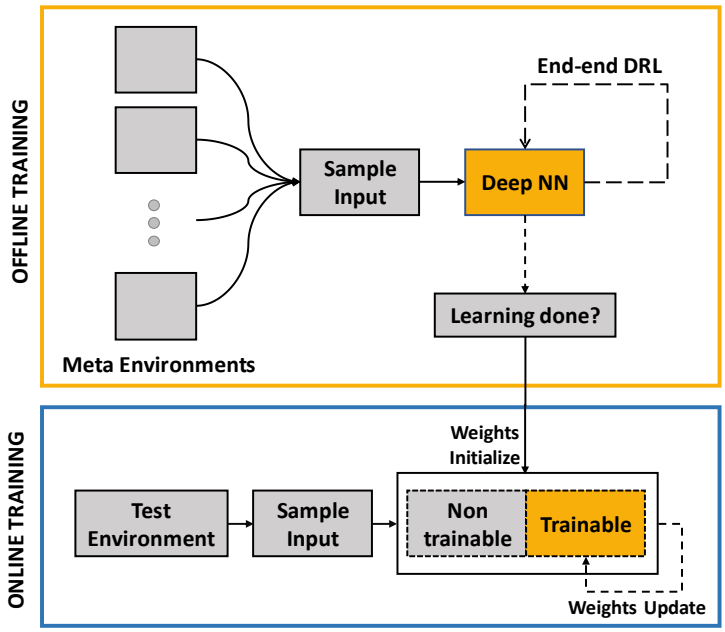


Figure 3.1: Block Diagram for the TL based Approach to DRL

3.2 TL based Proposed Approach

In this section, we discuss the Transfer Learning (TL) based approach targeting real-time and energy-efficient learning without compromising on the algorithmic performance. We propose a two-phase approach to the problems related to DRL which combines offline and online learning using Transfer Learning and fine-tuning. The idea is that if we train a NN for an RL application (say autonomous navigation) in a variety of indoor environments collectively, we can use this knowledge using TL training a smaller part of NN for similar applications in a similar (but different/unseen) test environment. The top-level block diagram of the approach can be seen in Figure 3.1 . In the Offline phase, one single network is trained on a set of training environments (called meta-environments) using DRL. These environments serve as a library of environments for the underlying problem. This offline training phase is carried out on the server (and not on edge nodes) where we assume no strict restriction on the compute engine. Once we have effectively trained a network on the meta environments collectively, we use these meta-weights as initialization during the online training phase. In the online training phase, a different test environment is used for

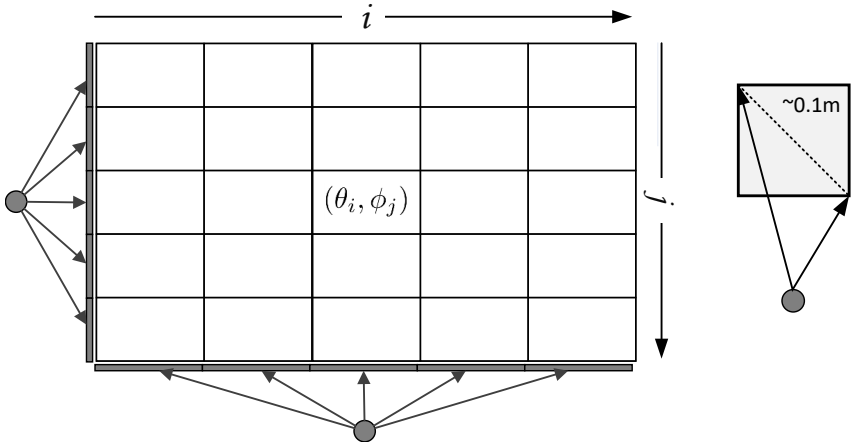


Figure 3.2: Perception based probabilistic Action Space A_s

training (fine-tuning). The training computations need to be carried out in the edge nodes (we don't implement anything on hardware, rather we provide the compute statistics and compare them with training the network end-to-end). In this phase, the training is only carried out on a part of the network. The network is divided into non-trainable and trainable parts and only the weights of the trainable part are updated. The segmentation of the network is a compromise between the performance (obstacle avoidance) and the number of training computations. Training the convolution (CONV) layer takes up much more computation as compared to that of the Fully Connected (FC) layer. Also, CONV layers capture the top-level features of the underlying problem such as edge detection, blurring, and sharpening and as we go deeper into the network, the features become more and more specific to the underlying problem. Hence including the CONV layers within the non-trainable part of the network makes much more sense. The trainable part of the network consists of the last few FC layers. The number of layers in the trainable part of the network is a parameter (called train type) that we vary during the experimentation. The variation of these train types is done by keeping the following two parameters in mind:

- **Similar performance:** For the reduced trainable size of the network, we ideally want it to perform similar to that of training the entire network (end-to-end training or e2e). The higher the similarity between the meta-environments and the test environment,

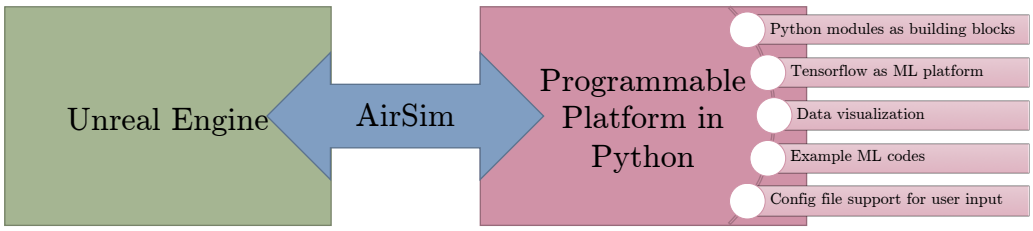


Figure 3.3: Python based training framework for drone related applications

the better the performance while training a smaller number of NN layers.

- **Reduced Training Computations:** With the reduced trainable weights, we want the training computations to be significantly lower than that of the e2e train type. This reduced computation will make the approach practical to be used on resource constraint edge nodes.

3.3 Python-based programming framework

To carry out experimentation of the proposed approach, a configurable programming framework PEDRA was developed Figure 3.3. The framework is developed in Python and is module-wise programmable. This framework is mainly targeted at goal-oriented RL problems for drones but can also be extended to other problems. The engine interfaces with the Unreal gaming engine using AirSim to create the complete platform. Unreal Engine [67] is used to create 3D realistic environments for the drones to be trained in. Different levels of detail are added to make the environments look as realistic as possible. These simulated environments are interfaced with the framework using AirSim [68]. AirSim is an open-source plugin developed by Microsoft that interfaces Unreal Engine with Python. It provides basic python functionalities controlling the sensory inputs and control signals of the drone. The custom framework is built onto the low-level python modules provided by AirSim creating higher-level python modules for the purpose of drone RL applications.

3.3.1 Perception-based probabilistic action space

Perception-based discrete action space A_s of size $N \times N$ is used. In this action space, the agent navigates by controlling the yaw and pitch expanding over all three coordinates. These angles are calculated by making use of the horizontal and vertical field-of-view (FOVs) of the front-facing camera. The camera image at time t, s_t is divided into $N \times N$ grid. Each window in the grid corresponds to an action in the action space. The action selection is simply the choice of the bin which is then transformed into velocity commands v_t for the drone. This velocity command results in moving along the line connecting the current position to the position where the window becomes the entire camera frame by r meters. Varying the pitch ϕ only results in the agent moving to one of the vertical bins while varying the yaw θ only moves the agent along the horizontal bins. Given the vertical and horizontal FOV (FOV_v and FOV_h) these angles are calculated as follows

$$\theta_i = \left(\frac{FOV_h}{N^2} \times (i - (N^2 - 1))/2) \right)$$

$$\phi_j = \left(\frac{FOV_v}{N^2} \times (j - (N^2 - 1)/2) \right)$$

where $i, j \in \{0, 1, \ldots, (N^2 - 1)/2\}$ is the (i, j) bin location as shown in Figure 3.2.

In all these actions, the agent moves forward by a constant distance of $r = 0.5m$. Moreover, the control associated with the action space is probabilistic. A uniform random noise $\epsilon \sim uniform(-b, b)$ is added to these deterministic yaw and pitch angles making them probabilistic and robust to slight control variations where $b = \frac{1}{15}$ is empirically selected. The maximum difference in final position under this probabilistic space for the same action is $\sim 0.1m$ and can be seen in Figure 3.2

3.3.2 Network Architecture

Deep Neural Network is used to map the state to their corresponding Q values based on a modified Alexnet architecture [53]. This architecture takes as input an RGB frame of size $227 \times 227 \times 3$ and outputs N^2 number of Q-values corresponding to each action in the action space. The network architecture can be seen in Figure 3.7. In order to help deep reinforcement learning converge better a dueling nature of the network [69] was used where we train two streams of FC network to estimate the state value function $V(s_t)$ and advantage function $A(s_t, a_t)$ separately which can be seen in the figure. The training approach used DoubleDQN [52] and Prioritized Experience Replay (PER) [70] to avoid the over-fitting nature of the Bellman Equation and aid faster learning respectively.

The complete network is trained during the offline phase while for the online phase a part of the network is used for training. Extra FC layers are added to the network to quantify the effect of training a certain number of layers in the online phase. A Parameter $train\ type$ is defined based on the number of layers that are trained. We evaluate the training for 3 different train types denoted by last$p$ and compared to the baseline of training the network end-to-end (e2e) where $p \in \{2, 3, 4\}$ denotes the number of FC layers trained from the end.

The idea behind these $train\ types$ is that training fewer layers will result in reduced computational cost. The details for these train types (number of weights, amount of Floating Point Operations FLOP) can be seen in Table 3.1. For modified Alexnet architecture, training the last$p$ layers for $p \in \{2, 3, 4\}$ results in a significant reduction in the number of floating-point operations required. This reduction in computations is directly co-related to the amount of energy required for training and is reported quantitatively in the subsection 3.5.2.

3.3.3 Simulated 3D environments:

We manually designed all the 3D indoor environments used for experimentation. These environments were built using an open-source gaming engine called Unreal Engine [67].

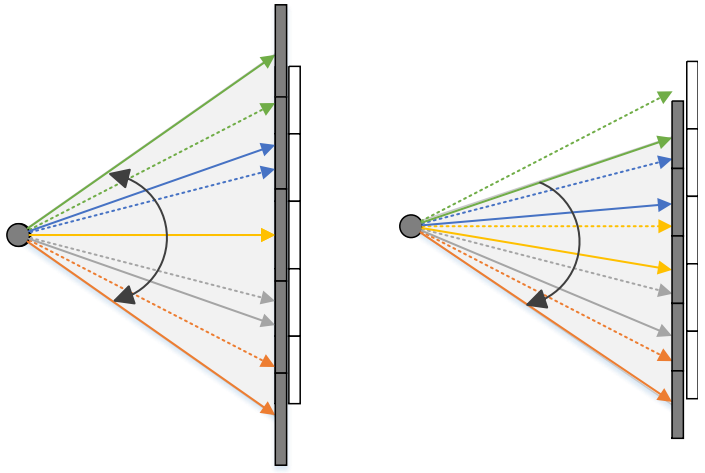


Figure 3.4: Variation in Action Space A_s. Right: Rotated Left: Dilated.

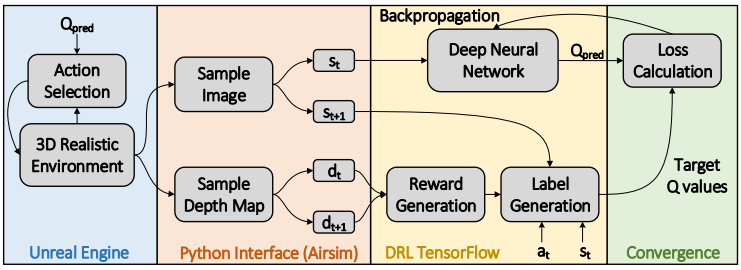


Figure 3.5: DRL Training Block Diagram

The designed environments contain a large variety of lighting conditions, hallway sizes, and structures such as long, broad, narrow, sharp turns, and circular hallways. Indoor furniture objects of various sizes were used to furnish these environments. The walls were textured with various patterns including metal, wood, marble, concrete, and wallpapers. These patterns were selected randomly from a pool of 40 textures to create a diverse data set. Learning a network on this wide variety of indoor environments will help us generalize it to other rendered environments. The more the variation of parameters in the simulation, the better the network is able to generalize the problem. The floor plan and screenshots of the 8 meta-environments can be seen on Figure 3.8.

Algorithm 3: OFFLINE TRAINING PHASE ALGORITHM

Input: Set of N meta environments: $\mathcal{E}_{meta} = \{\mathcal{E}_0, \mathcal{E}_1, ..., \mathcal{E}_N, \}$
Output: Weights of neural network θ_{meta}
Initialization: Behaviour network: $Q_\theta(s) = \mathcal{N}(s; \theta)$, Target network: $Q_{\theta'}(s) = \mathcal{N}(s; \theta')$, n_{target}: Target network update interval, n_{batch}: mini-batch size for training, n_{train}: Train Interval, $\mathcal{D}_{replay}$, $env = 0$, m : Environment switch interval
for $t \in \{1, 2, 3, ..., max\_steps\}$ **do**
  if $mod(t, m) = 0$ **then**
    $saved\_state[env] \leftarrow (s_t, p_t)$
    $env \leftarrow mod((env + 1), N)$
    $(s_t, p_t) \leftarrow saved\_state[env]$
    $\mathcal{E}_{current} \leftarrow \mathcal{E}_{env}$
    $position\_agent(\mathcal{E}_{current}, p_t)$
  else
    $s_t \leftarrow get\_state(\mathcal{E}_{current}, p_t)$
  end
  Sample an action a_t from current policy using ϵ-greedy
  $p_{t+1} \leftarrow move\_agent(\mathcal{E}_{current}, p_t, a_t)$
  $s_{t+1} \leftarrow get\_state(\mathcal{E}_{current}, p_{t+1})$
  $r_t \leftarrow get\_reward(s_t, a_t, s_{t+1}, p_{t+1})$
  Store the tuple (s_t, a_t, s_{t+1}, r_t) in $\mathcal{D}_{replay}$
  if $mod(t, n_{train}) = 0$ **then**
    Sample a mini-batch of size n_{batch} from $\mathcal{D}_{replay}$ Train the Behaviour network: $Q_\theta(s) = \mathcal{N}(s; \theta)$
  end
  if $mod(t, n_{target}) = 0$ **then**
    $\theta' \leftarrow \theta$
  end
end
$\theta_{meta} \leftarrow \theta$

3.4 Experimentation

The idea is to show that once the network was trained on meta environments, this knowledge can be used to help train another network for a similar but different problem. The similarity of the problem is kept by having the same object, i.e. autonomous navigation, while the 'different' part is achieved by changing/varying the environment and action space. This is done to show that this learning approach is robust to variation in environment and agent's control dynamics. The complete training block diagram can be seen in Figure 3.5.

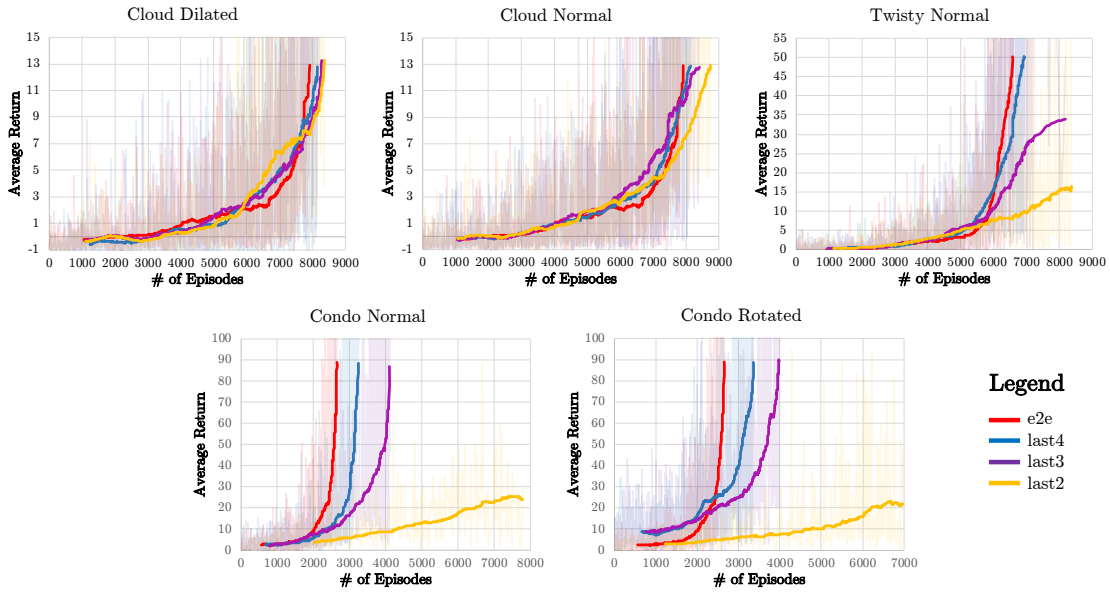


Figure 3.6: Return across environment and action space combination

Table 3.1: Weights and FLOP for each train type

Train Type	# of weights	FLOP	% weights	% FLOP
e2e	48,858,522	5.16G	100%	100%
last4	7,358,490	7.35M	15.06%	0.14%
last3	3,162,138	3.15M	6.47%	0.06%
last2	1,062,938	1.06M	2.17%	0.02%

3.4.1 Environmental Variation

Environmental variation was carried out by designing 3 test environments (named Cloud, Condo, and Twisty) with variation in the floor plan, lighting, and textures as that of used in the meta environments. The floor plans and snapshots at different locations of these test environments can be seen in the Figure 3.9. These environments were designed with a varying degree of similarity to the environments in the used for meta-training and will be discussed in the next section.

3.4.2 Action Space Variation

Action space variation was carried out by defining 2 other action spaces along with the one used during the meta training phase. The actual action space was dilated and rotated to

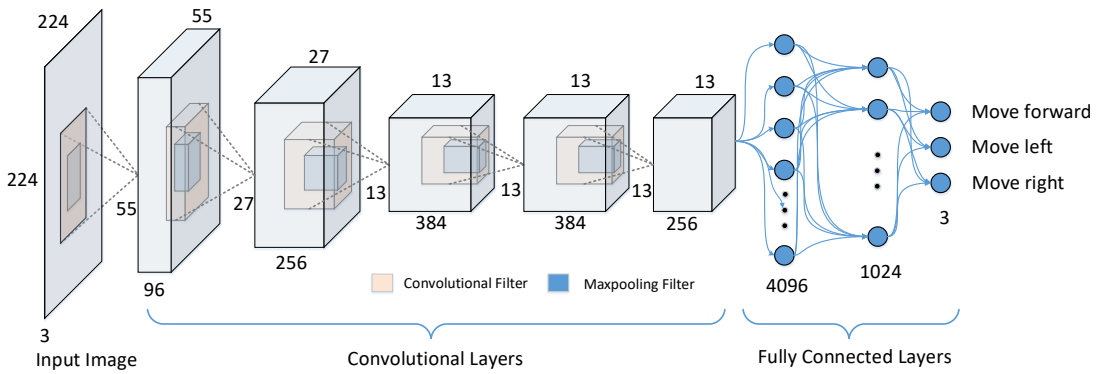


Figure 3.7: Modified Alexnet used for training

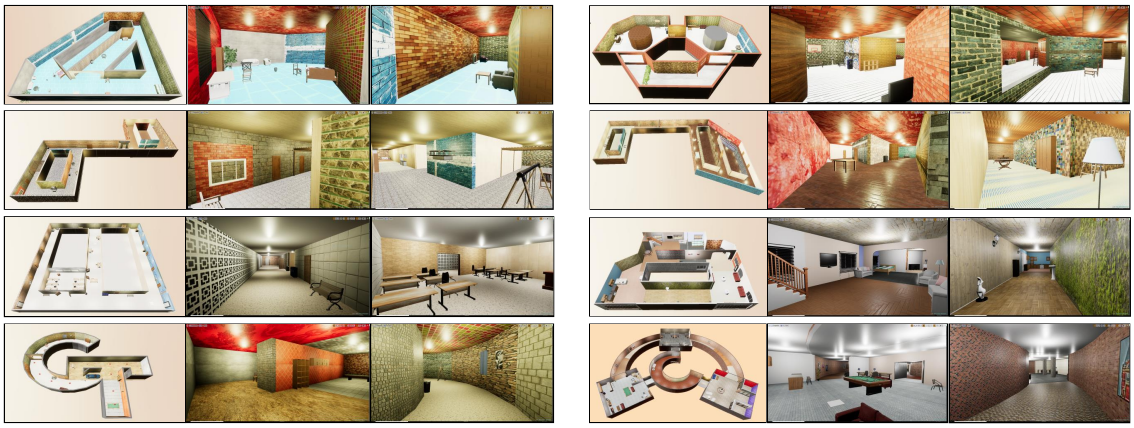


Figure 3.8: 3D floor plan and screen-shots of the 8 meta environments used for offline training phase.

generate two other action spaces. The explanation of these action spaces can be seen in Figure 3.4. The dilated action space was created by dilating the yaw and pitch angles in the original action space by 20%, while the rotated action space rotates the original action space by 25% for both pitch and yaw. Both of these action spaces were made probabilistic by introducing noise in the angles (pitch and yaw) as explained in the previous section.

3.5 Experimental Results

In this section, we evaluate the proposed approach and quantify the algorithmic performance and computational cost for each train type across different test environments. Experimentation was carried out on a workstation with GTX1080 GPU. As mentioned in the

Figure 3.9: 3D floor plan and screen-shots of the 3 test environments used for online training phase. (from left to right) Cloud, Condo and Twisty

		Environment variation		
		Cloud	Condo	Twisty
Action Variation	Normal	■	■	■
	Rotated		■	
	Dilated	■		
Each block contains 4 RL runs i.e. e2e, last4, last3, last2				

Figure 3.10: Training combination used across the environment and action space variation

previous section, a list of 20 experimentation was carried out by varying the environment and action space. The list of combination used during experimentation is shown in Figure 3.10

For each of these combinations, the agent was initialized at three different initial positions randomly chosen prior to learning. A dueling network was learned using DDQN and PER. The network was first trained end-to-end updating all the weights of the network for 150,000 steps and the return was recorded. The algorithm used for the offline training phase can be seen in algorithm 3. This return serves as a baseline setting a threshold for subsequent train types (last4, last3, and last2). For these train types, the network was trained for either at most 300,000 steps or until the return match that of the e2e train type.

Figure 3.11: Mean Safe Flight (MSF) across different environment for different action spaces.

3.5.1 Algorithmic Performance

The return graph for all these combinations has been plotted in Figure 3.6. The return graph reported/plotted is the moving average of the actual return graph to make it more meaningful. It can be seen that in all the cases, *train type* last4, and for some cases others, were able to match the return obtained for the train type e2e. It should be noted that variations in action space didn't bar the network to achieve the required return. The only difference that it made was the time/steps required to achieve that return. It took slightly longer to achieve the desired return.

Test environment 1 - Cloud

This environment had a smooth floor plan (no sharp edges) and all the wall textures used in this environment were chosen from the 40-texture pool used in the construction of the meta environment. The Amount of learning transferred from meta environments to this test environment should be significant due to its greater similarity to meta environments. This can be seen in the return graph for this environment as shown in Figure 3.6. Not only did all the train types were able to reach the desired return value, but they also did it in almost equal number/amount of iterations/time.

Table 3.2: Mean Safe Flight (MSF)

Mean Safe Flight (m)						
Env	A_s	**e2e**	**last4**	**last3**	**last2**	**meta**
Cloud	Normal	1245.7	1209.0	1206.5	1187.5	110.0
	Dilated	1245.7	1203.0	1197.0	1093.0	110.0
Condo	Normal	1218.5	1196.6	1175.1	512.1	77.8
	Rotated	1218.5	1153.4	1118.0	425.0	77.8
Twisty	Normal	580.7	573.2	468.4	248.0	68.8

Test Environment 2 - Condo

The floor plan of this environment had turns similar to that of the meta environment. 75% of the textures used for the walls were chosen from the 40 textures pool used during the design of the meta environments. The rest of the 25% textures were the ones that were never used in the meta environments. The idea is to evaluate the robustness of the approach to variation in the environment characteristics. The idea is to evaluate the performance of the approach to an unseen textured environment. The return graph for different train types can be seen in Figure 3.6. It can be seen that except for the train type last2, all the other train types were able to achieve the desired return value. Since this environment has lesser similarity with meta environment as compared to that of Cloud environment, the train type last4 and last3 took longer to achieve the desired return value.

Test Environment 3 - Twisty

Half of the textured used in this environment was new and had never used in the design of meta environments. The floor plan has sharp turns and narrower hallways as compared to other environments. Only train type last4 was able to achieve the desired return threshold, while the last 3 performed better than the last 2. The respective return graph can be seen in Figure 3.6.

Mean Safe Flight (MSF) was used to meaningfully quantify the performance of the learned networks in the respective environment. MSF is the average distance traveled by the

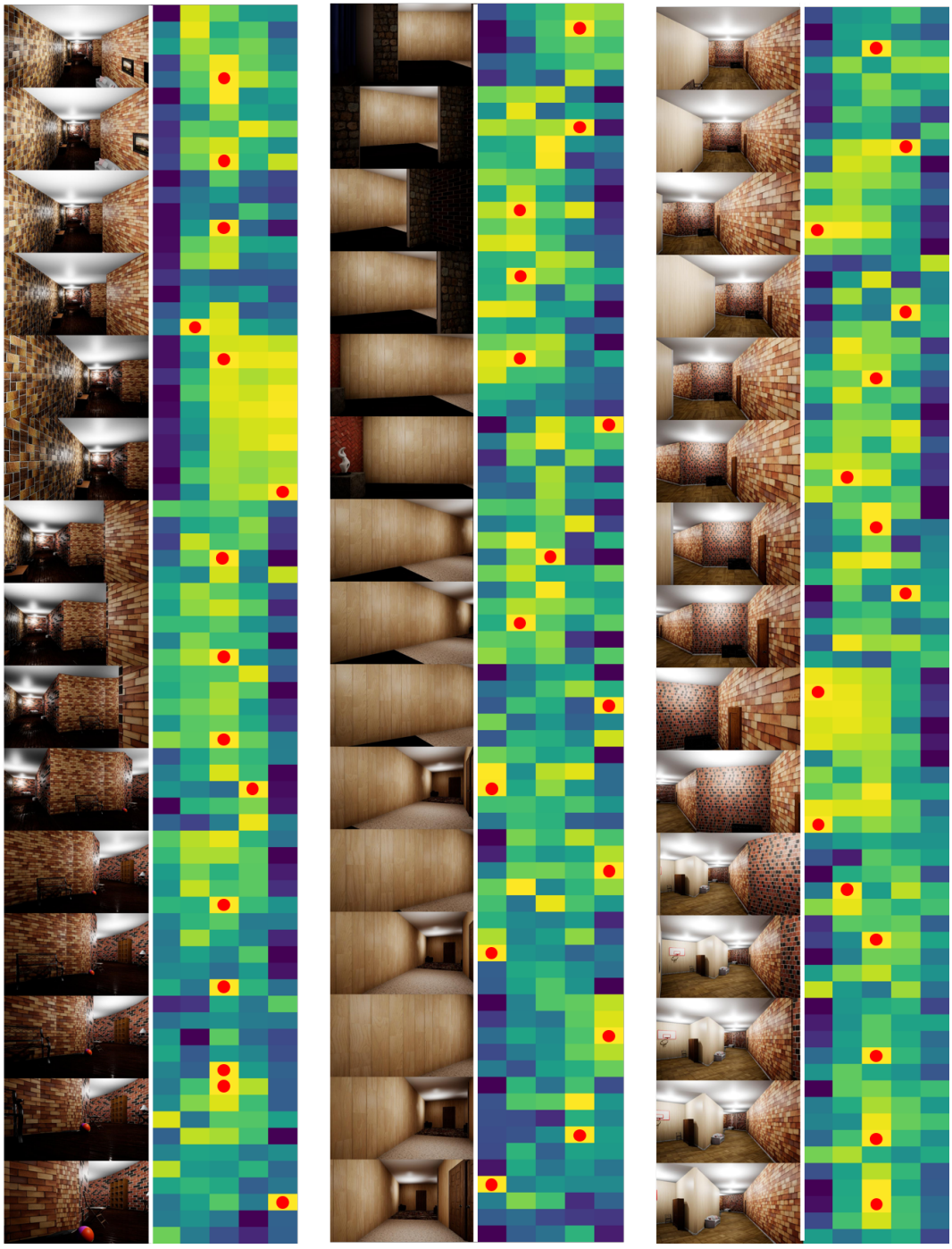


Figure 3.12: Images captured from front facing camera of the drone during flight in simulated environments. On the right of each block is the action space probability where blue corresponds to lower and yellow higher probability. The red dot corresponds to the action taken. From left to right: Cloud, Condo and Twisty environment

Table 3.3: GPU parameters for different train types

Train Type	Runtime(s)	DtoD memCpy (MB)	GPU Mem (MB)	GPU Load (%)	Energy/iter (J)
e2e	40.59617	586.3	4389.0	0.40188	5.87
last4	24.01384	254.3	3364.0	0.21432	1.85
last3	23.17413	220.7	3362.0	0.20457	1.71
last2	22.67000	203.9	3298.0	0.19234	1.57

agent, in meters, before a collision. For each of the learning combinations, the network was initialized with the learned weights and the agent was initialized randomly at 10 different locations within the environment. In order to have a fair comparison, the agent was placed exactly the same way (in terms of position and orientation) across all the train types. In each of the cases, the distance traveled by the agent before the collision was recorded and averaged out to generate the MSF. These actual MSF values can be seen in the Table 3.2 and the normalized MSF value for each environment is plotted in Figure 3.11. The rightmost column 'meta' or 'no'train' shows the MSF values achieved by the network initialized with meta-weights without fine-tuning. It can be seen that for all the cases, the MSF achieved by the train type last4 is at least 97% that of achieved by end-to-end training. MSF achieved by all the train types co-relates with their return values.

Figure 3.12 shows the images captured from the front-facing camera of the drone during a flight across the three different **simulated** test environments. For each environment, the RGB image of the camera (on the left) and the 5×5 network predicted action space (on the right) have been shown. Each bin in the predicted action space represents the normalized Q values (across all the predictions). The darker (blue) bins correspond to smaller values while lighter (yellow) corresponds to higher Q values. Moving in the direction of darker bins will increase the probability of collision.

3.5.2 Computational Cost

To measure the resources used during training, for each of the train types, a set of GPU parameters were recorded. These computational parameters were collected using NVIDIA's profiling tools (nvidia-smi [71] and nvprof [72]) and include

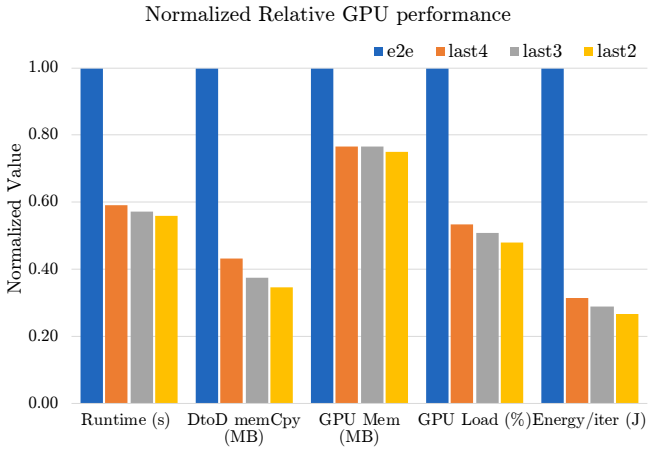


Figure 3.13: GPU parameters for the 4 different train types

- **Runtime**: Time in seconds taken to train the neural network for K iterations
- **DtoDMemcpy**: Amount of data transferred (in MBs) within the GPU cores
- **GPU Mem**: Amount of GPU Memory used
- **GPU Load**: Power consumption of GPU in Watts
- **Energy/iter**: Energy consumption per training iteration

Runtime and GPU load corresponds to latency and power required for training, while DtoD-Memcpy and GPU Mem govern the hardware resources required. These parameters give a quantitative way of understanding how these different train types directly affect the edge node resources. In order to calculate these parameters, for each train type, the neural network was trained for $K = 500$ number of iterations on a collected dataset. These GPU parameters have been tabulated in Table 3.3 and their normalized values have been plotted in Figure 3.13. The energy per iteration in the table is calculated from the power consumed by the GPU, total run-time, and the number of iterations.

$$Energy/iter = \frac{GPU\ load(\%) \times max\ GPU\ load \times Runtime}{number\ of\ iterations}$$

It can be seen that for all the train types the time required to train the network (latency) was

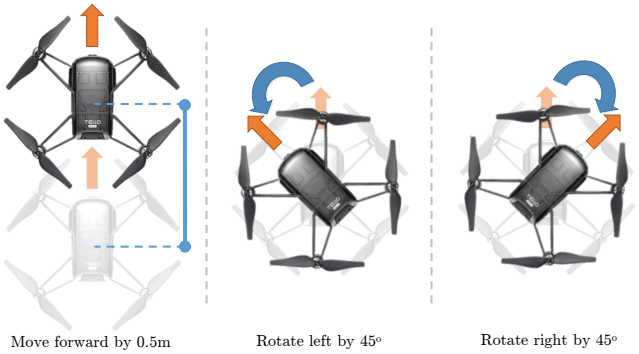


Figure 3.14: Action space of Tello drone for real environment

reduced to less than 60% as compared to that of e2e while reducing the energy consumption to less than 30%. The reduced latency directly dictates the speed of the drone during training. For a given speed of the drone, the corresponding distance traveled between two sequentially acquired frames, and the drone distance threshold for obstacles (a measure of clutter in the environment), we can calculate the minimum number of Frames per Second (FPS) required for collision avoidance. For a drone to have a higher speed, it needs to be able to process more frames in a given amount of time (i.e. support higher FPS). The drone will only be able to support that speed if the underlying computational system can process the dictated FPS (which is inverse of the per-frame latency). So, the maximum speed of the drone will be limited by the latency of the system. Hence the latency improvement of last2 vs e2e in Figure 3.13 directly corresponds to an improvement of maximum supported theoretical speed (based purely on the training pass and ignoring other latency sources) of about 1.8 times from e2e to last2. Using the lower train types not only reduces the latency but also requires less operating power. Since it was reported in Figure 3.11 that the algorithmic performance (in terms of MSF) for these different train types was comparable to e2e learning, reduced hardware, power, and time requirement makes it favorable to be implemented on edge nodes.

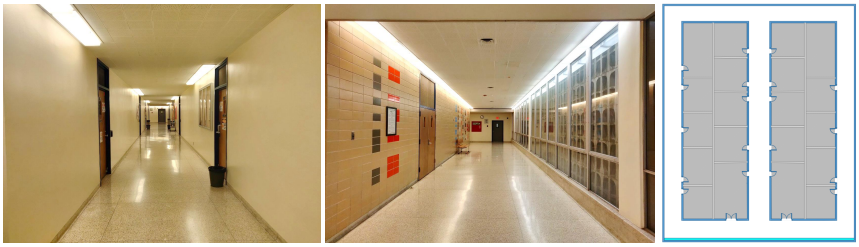


Figure 3.15: Snapshots and the layout of the Hallway arena used as test real environment

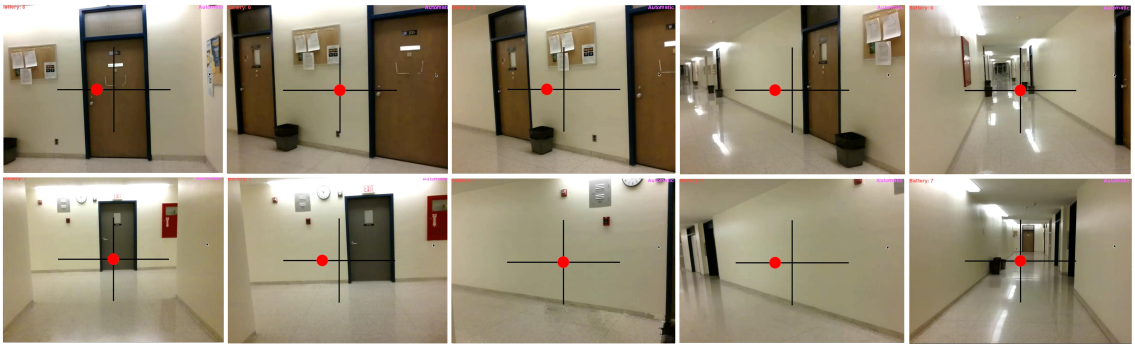


Figure 3.16: Action predictions by the network for the Hallway environment

3.5.3 Experimental Verification with DJI Drone in Real Environment

In this section, the result of implementing the proposed approach on a real drone in a real environment is reported and is compared with other baseline algorithms. A low-cost DJI Tello drone was used for this real-time experimentation. DJI Tello does not have the computational power to carry out the required processing onboard. Hence, a workstation/cloud equipped with a Core i7 processor and GTX1080 GPU was used for training. TensorFlow was used as the ML platform to carry out the neural network computation on the workstation.

For the proposed approach, the offline training was carried out on the same set of simulated meta-environments (Figure 3.9) and modified AlexNet network (Figure 3.7) as discussed in the previous section. The action space, however, was modified to contain only three actions. These actions include going forward by 0.5m, rotating clockwise by 45 degrees, and rotating counterclockwise by 45 degrees, and can be seen in Figure 3.14. The action space did not include any actions that correspond to changing the drone altitude.

Once the network was trained for the three-action action space on the simulated meta-environments, the learned weights were used as initializers for the network to be trained in a real environment. For this purpose, a hallway environment of an engineering building was used that contains glass walls and corridors $\sim 1.5m$ wide and can be seen in Figure 3.15. Using the baseline deep reinforcement learning algorithm in a real environment is time-consuming. Hence the approach discussed in [73] was used. Using this approach, an expert user collects a set of data points in the real environment. These expert data points are made a mandatory part of the experience replay from which the data points are sampled for training. Moreover, data-aggregation techniques are used when the drone virtually crashes to aid the data collection. Only the last two layers of the network were updated during training, while the weights in the rest of the layers were kept static.

Once the network was trained for the last 2 layers, the drone was placed at different initial positions and the performance of the network was observed. MSF was used as the performance metric.Figure 3.16 shows the control actions predicted by the network for the given camera frames. The performance of the proposed approach is also compared with the following baseline algorithms.

- **Straight-line (SL) controller**: Always predicts moving forward, providing a qualitative idea of the complexity of the arena [38]
- **Left-Right-Straight (LRS) controller** : Supervised approach to classify images with respect to the actions required to be taken [48]
- **Self-supervised (SS) controller**: 11,500 videos of various indoor environments are used to train a network to classify images as safe or crash. A handcrafted algorithm is designed to take suitable actions avoiding obstacles [45].

All of these baselines have the same three-actions action space. The MSF in meters for the proposed and baseline algorithms in the Hallway environment can be seen in Figure 3.17. It can be seen that the proposed approach (DRLwitTL) performs better as com-

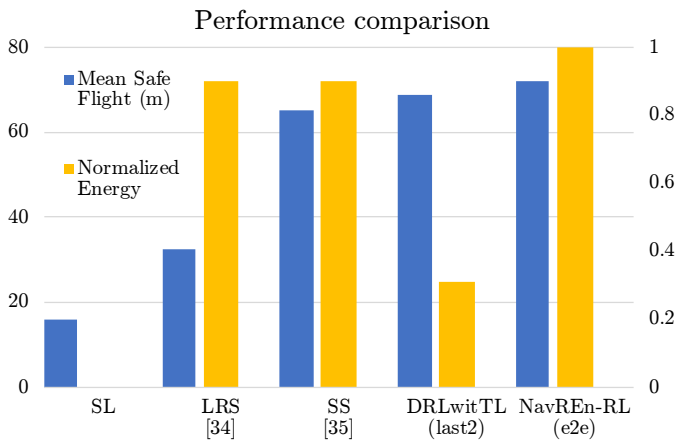


Figure 3.17: Performance comparison across baseline algorithms

pared to the other baseline algorithms. Moreover, DRLwithTL performs almost similar to that of NAVREN-RL which corresponds to the e2e *train type* i.e. training the entire network. The important point to note is the amount of energy used to carry out the proposed and baseline algorithms. It can be seen that the energy consumption was reduced by a factor of x3.

3.6 Summary

This chapter discusses a Transfer learning approach to reduce the number of resources required to train a deep neural network for RL problem by training the network on a set of rich and diverse meta environments, transferring the domain knowledge to test environments, and training the last few fully connected layers only. The algorithmic performance of this network measured in terms of Mean Safe Flight was similar to training the network end-to-end while reducing the latency and energy consumption by 1.8 and 3.7 times respectively. The reduction in these parameters can make it possible for DRL training to implemented resource-constrained edge nodes. Moreover, the approach was tested in a real environment using a low-cost drone and showed similar performance when compared across different baselines.

Even though we get improvements in terms of energy and latency numbers, we can

only do enough when it comes to using algorithmic approaches in improving the latency and energy efficiency of an RL system. The two parts of the network, trainable and non-trainable parts, can be mapped onto different memory technologies to further improves energy efficiency [74, 75]. The non-trainable part will not be updated and hence does not require writing into that often. This part can be mapped onto a low standby leakage Spin Transfer Torque (STT) RAM which is cheap but has moderate to slow write speeds. On the other hand, the smaller trainable part of the network can be mapped onto a fast, expensive but energy/latency efficient on-die SRAM.

CHAPTER 4

ADDRESSING MULTI-AGENT SYSTEM - MTRL

Up until now, we discussed various approaches to improve the energy efficiency of a single-agent RL system by modifying the vanilla RL algorithm and using transfer learning. We used transfer learning to learn global features for the underlying network model from simulation and learning the local features on the actual test environment. This is one way of using TL to share knowledge. Transfer learning can be further used to share knowledge across a multi-agent multi-task distributed RL system. Sharing of knowledge across different agents can help each agent learn its task faster, hence saving up on the extra computations required [76]. Such distributed systems can help converge the RL algorithm faster but are also vulnerable to adversarial attacks that can compromise the performance of the RL system.

In this chapter, we briefly explore distributing the compute across multiple agents with their own assigned tasks using multi-task federated RL (MT-FedRL) and discuss in detail how such algorithms are vulnerable to adversarial attacks. We also propose modifications to existing MT-FedRL algorithms to make them immune to such attacks.

4.1 Introduction

Just like humans, ML has also been benefiting from the shared representations across different tasks that need to be implemented. The idea behind multi-task RL is to learn a shared representation, say a neural network model, for different tasks at hand [77, 78, 79, 80, 81, 82, 83]. MTRL assumes that the underlying tasks are co-related and can be learned jointly. Learning multiple tasks, instead of learning a single task, has several advantages. A common representation of the learned tasks saves up memory, relaxing the memory requirements. Furthermore, it has also been shown that learning a common representation

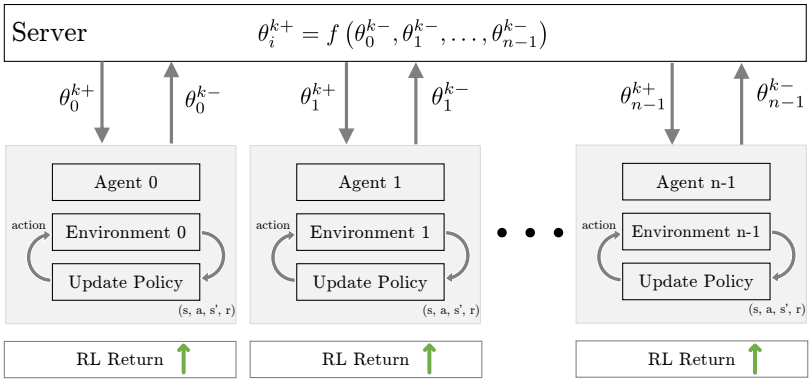


Figure 4.1: Federated RL - The idea is to learn a common unified policy without sharing the local training data that works good enough for all the environments

across tasks is faster than learning a single task [76]. The latter benefit indirectly translates to improved energy efficiency. The faster the convergence of the RL algorithm, the lesser the amount of energy consumed.

Distributed algorithms have been studied to take advantage of these distributed compute agents. The conventional methods consist of using these IoT as data collectors and then using a centralized server to train a network on the collected data. Federated Learning, introduced by Google [84, 85, 86] is a distributed approach to machine learning tasks enabling model training on large sets of decentralized data by individual agents. The key idea behind federated learning is to preserve the privacy of the data to the local node responsible for generating it. The training data is assumed to be local only, the agents, however, can share the model parameter that is learned.

While ML algorithms have proven to provide superior accuracy over conventional methods, they pose a threat from adversarial manipulations. Adversaries can use a variety of attack models to manipulate the model either in the training or the inference phase leading to decreased accuracy or poor policies. Common attack methods include data-poisoning and model poisoning where the adversary tries to manipulate the input data or directly the learned model respectively. In this chapter, we propose and analyze a model-poisoning attack for the Multi-task Federated RL (MT-FedRL) problem and modify the

conventional Federated RL approach to provide protection from model poisoning attacks.

The contributions of this chapter are as follows

- We carry out a detailed study on the multi-task federated RL (MT-FedRL) with model-poisoning adversaries on medium and large size problems of grid-world (GridWorld) and drone autonomous navigation(AutoNav).
- We argue that the general adversarial methods are not good enough to create an effective attack on MT-FedRL, and propose a model-poisoning attack methodology *AdAMInG* based on minimizing the information gain during the MT-FedRL training.
- Finally, we address the adversarial attack issue by proposing a modification to the general FedRL algorithm, `ComA-FedRL`, that works equally well with and without adversaries.

4.2 Related Work

The effects of adversaries in Machine learning algorithms were first discovered in [87] where it was observed that a small l_p norm perturbation to the input of a trained classifier model resulted in confidently misclassifying the input. These l_p norm perturbations were visually imperceptible to humans. The adversary here acts in the form of specifically creating adversarial inputs to produce erroneous outputs to a learned model [88, 89, 90, 91, 92, 93]. For supervised learning problems, such as a classification task, where the network model has already been trained, attacking the input is the most probable choice for an adversary to attack through. In RL, there is no clear boundary between the training and test phase. The adversary can act either in the form of data-poisoning attacks, such as creating adversarial examples[94, 95], or can directly attack the underlying learned policy [96, 97, 98, 99] either in terms of malicious falsification of reward signals, or estimating the RL dynamics from a batch data set and poisoning the policy. Authors in [100], try to attack an RL agent by selecting an adversarial policy acting in a multi-agent environment as a result of

creating observations that are adversarial in nature. Their results on a two-player zero-sum game show that an adversarial agent can be trained to interact with the victim winning reliably against it. In federated RL, alongside the data-poisoning and policy-poisoning attacks, we also have to worry about the model-poisoning attacks. Since we have more than one learning agent, a complete agent can take up the role of an adversary. The adversarial agent can feed in false data to purposely corrupt the global model. In model poisoning attacks the adversary, instead of poisoning the input, tries to adversely modify the learned model parameters directly by feeding false information purposely poisoning the global model [101, 102]. Since federated learning uses an average operator to merge the local model parameters learned by individual agents, such attacks can severely affect the performance of the global model.

Adversarial training can be used to mitigate the effects of such adversaries. [103] showed that the classification model can be made much robust against the adversarial examples by feature de-noising. The robustness of RL policies has also been analyzed by the adversarial training [104, 105, 106, 107]. [104, 107] show that the data-poisoning can be made a part of RL training to learn more robust policies. They feed perturbed observations during RL training for the trained policy to be more robust to dynamically changing conditions during test time. [108] shows that the data-poisoning attacks in federated learning can be resolved by modifying the federated aggregation operator based on induced ordered weighted averaging operators [109] and filtering out possible adversaries. To the best of our knowledge, there is no detailed research carried out on MT-FedRL in the presence of adversaries. In this chapter, we address the effects of model poisoning attacks on the MT-FedRL problem.

4.3 Multi-task Federated Reinforcement Learning (MT-FedRL)

We consider a Multi-task Federated Reinforcement Learning (MT-FedRL) problem with n number of agents. Each agent operates in its own environment which can be characterized

by a different Markov decision process (MDP). Each agent only acts and makes observations in its own environment. The goal of MT-FedRL is to learn a unified policy, which is jointly optimal across all of the n environments. Each agent shares its information with a centralized server. The state and action spaces do not need to be the same in each of these n environments. If the state spaces are disjoint across environments, the joint problem decouples into a set of n independent problems. Communicating information in the case of N-independent problems does not help.

We consider policy gradient methods for RL. The MDP at each agent i can be described by the tuple $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{P}_i, \mathcal{R}_i, \gamma_i)$ where $\mathcal{S}_i$ is the state space, $\mathcal{A}_i$ is the action space, $\mathcal{P}_i$ is the MDP transition probabilities, $\mathcal{R}_i : \mathcal{S}_i \times \mathcal{A}_i \to \mathbb{R}$ is the reward function, and $\gamma_i \in (0, 1)$ is the discount factor.

Let V_i^π be the value function, induced by the policy π, at the state s in the i-th environment, then we have

$$V_i^\pi(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma_i^k \mathcal{R}_i(s_i^k, a_i^k) \mid s_i^0 = s\right],$$
$$a_i^k \sim \pi(\cdot|s_i^k). \tag{4.1}$$

We denote by ρ_i the initial state distribution over the action space of i-th environment., The goal of the MT-FedRL problem is to find a unified policy π^* that maximizes the sum of long-term discounted return for all the environment i i.e.

$$\max_\pi V(\pi; \boldsymbol{\rho}) \triangleq \sum_{i=0}^{n-1} \mathbb{E}_{s_i \sim \rho_i} V_i^\pi(s_i), \quad \boldsymbol{\rho} = \begin{bmatrix} \rho_0 \\ \vdots \\ \rho_{n-1} \end{bmatrix} \tag{4.2}$$

Solving the above equation will yield a unified π^* resulting in a balanced performance

across all the environments.

We use the parameter θ to model the family of policies $\pi_\theta(a|s)$, considering both the tabular method (for simpler problems) and neural network-based function approximation (for complex problems). The goal of the MT-FedRL problem then is to find θ^* satisfying

$$\theta^* = \arg\max_{\theta} V(\theta; \boldsymbol{\rho}) \triangleq \sum_{i=0}^{n-1} \mathbb{E}_{s_i \sim \rho_i} V_i^{\pi_\theta}(s_i). \tag{4.3}$$

In tabular method, gradient ascent methods are utilized to solve (Equation 4.2) over a set of randomized stationary policies $\{\pi_\theta : \theta \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}\}$, where θ uses the softmax parameterization

$$\pi_\theta(a \mid s) = \frac{\exp(\theta_{s,a})}{\sum_{a' \in \mathcal{A}} \exp(\theta_{s,a'})}. \tag{4.4}$$

For a simpler problem where the size of state-space and action-space is limited, this table is easier to maintain. For more complex problems with a larger state/action space, usually neural network-based function approximation $\{\pi_\theta : \mathcal{S} \to \mathcal{A}\}$ is used, where θ are the trainable weights of a pre-defined neural network structure.

One approach to solving this unified-policy problem is by sharing the data $\mathcal{M}_i$ observed by each agent in its environment to a centralized server. The centralised server then can train a single policy parameter θ based on the collective data $\mathcal{M} = \cup_{i=0}^{n-1} \mathcal{M}_i$. This, however, comes with the cost of reduced privacy as the agent needs to share its data with the server. In MT-FedRL, however, the data tuple $\mathcal{M}_i$ is not shared with the server due to privacy concerns. The data remains at the local agent and instead, the policy parameter θ_i is shared with the server. Each agent i utilizes its locally available data $\mathcal{D}_i$ to train the policy parameter θ_i by maximizing its own local value function V_i^π through SGD. We assume policy gradient methods for RL training. After the completion of each episode k, the agents share their policy parameter θ_i^{k-} with a centralized server. The server carries out a smoothing average and generates N new sets of parameters θ_i^{k+}, one for each agent, using

the following expression.

Algorithm 4: Multi-task Federated Reinforcement Learning (MT-FedRL) with smoothing average

Initialization: $\theta_i^{0+} \in \mathbb{R}^d$, step sizes δ^k, smoothing average threshold iteration t

`%Server Executes`

for *k=1,2,3,...* **do**

Calculate smoothing average parameters

$$\alpha^k = \frac{1}{n}\min(1, k/t), \quad \beta^k = \frac{1-\alpha^k}{n-1}$$

for *each agent i* ***in parallel*** **do**

Receive updated policy parameter from clients

$\theta_i^{(k+1)-} \leftarrow$ `ClientUpdate` $\left(i, \theta_i^{k+}\right)$

end

for *each agent i* **do**

Policy update:

$$\theta_i^{(k+1)+} = \alpha^k \theta_i^{(k+1)-} + \beta^k \sum_{i \neq j} \theta_j^{(k+1)-}$$

Send updated policy parameter $\theta_i^{(k+1)+}$ back to client i

end

end

Function `ClientUpdate(`i, θ`)`**:**

1) Compute the gradient of the local value function $\frac{\partial V_i^{\pi_\theta}(\rho_i)}{\partial \theta_{s_i,a_i}}$ based on the local data

2) Update the policy parameter

$$\theta^- = \theta + \delta^k \frac{\partial V_i^{\pi_\theta}(\rho_i)}{\partial \theta_{s_i,a_i}} \tag{4.5}$$

return θ^-

$$\theta_i^{k+} = \alpha^k \theta_i^{k-} + \beta^k \sum_{j \neq i} \theta_j^{k-} \tag{4.6}$$

where $\alpha^k, \beta^k = \frac{1-\alpha}{n-1} \in (0, 1)$ are non-negative smoothing average weights. The goal of this

smoothing average is to achieve a consensus among the agents' parameters, i.e.

$$\lim_{k\to\infty} \theta_i^{k+} \to \theta^* \quad \forall i \in \{0, n-1\} \tag{4.7}$$

As the training proceeds, the smoothing average constants converge to $\alpha^k, \beta^k \to \frac{1}{n}$. The conditions on α^k, β^k to guarantee the convergence of algorithm 4 can be found in [83]. The complete algorithm of multi-task federated RL can be found in algorithm 4

4.4 MT-FedRL with adversaries

MT-FedRL has proven to converge to a unified policy that performs jointly optimal on each environment [83]. This jointly optimal policy yields near-optimal policies when evaluated on each environment if the agents' goals are positively correlated. If the agent's goals are not positively correlated, the unified policy might not result in a near-optimal policy for individual environments. This is exactly what happens to MT-FedRL in the presence of an adversary.

Let $\mathcal{L}$ denote the set of adversarial agents in a $n - agent\ MTFedRL$ problem. The smoothing average at the server can be decomposed based on the adversarial and non-adversarial agent as follows

$$\theta_i^{k+} = \alpha^k \theta_i^{k-} + \beta^k \sum_{j\neq i, j\notin\mathcal{L}} \theta_j^{k-} + \beta^k \sum_{l\in\mathcal{L}} \theta_l^{k-} \tag{4.8}$$

where $i \notin \mathcal{L}$. θ_i^{k+} is the updated policy parameter for agent i calculated by the server at iteration k. This update incorporates the knowledge from other environments and as the training proceeds, these updated policy parameters for all the agents converge to a unified parameter θ^*. In a non-adversarial MT-FedRL problem, this unified policy parameter ends up achieving a policy that maximizes the sum of discounted returns for all the environments. In an adversarial MT-FedRL problem, the goal of the adversarial agent is to prevent the MT-FedRL from achieving this unified θ^* by purposely providing an adversarial policy

parameter θ_l^{k-}.

Parameters that effect learning: Using gradient ascent, each agent updates its own set of policy parameter locally according to the following equation,

$$\theta_i^{k-} = \theta_i^{(k-1)+} + \delta_i \nabla_{\theta_i} V_i^{\pi_{\theta_i}}(\rho_i) \tag{4.9}$$

where δ_i is the learning rate for agent i. Using Equation 4.9 in the smoothing average Equation 4.6 yields

$$\begin{aligned} \theta_i^{k+} = &\left(\alpha^k \theta_i^{(k-1)+} + \beta^k \sum_{j \neq i, j \notin \mathcal{L}} \theta_j^{(k-1)+} \right) + \\ &\left(\alpha^k \delta_i \nabla_{\theta_i} V_i^{\pi_{\theta_i}}(\rho_i) + \beta^k \sum_{j \neq i, j \notin \mathcal{L}} \delta_j \nabla_{\theta_j} V_j^{\pi_{\theta_j}}(\rho_j) \right) + \\ &\left(\beta^k \sum_{l \in \mathcal{L}} \theta_l^{k-} \right) \end{aligned} \tag{4.10}$$

The server update of the policy parameter can be decomposed into three parts.

- The weighted sum of the previous set of policy parameters $\theta_i^{(k-1)+}$ shared by the server with the respective agents.
- The agent's local update, which tries to shift the policy parameter distribution towards the goal direction.
- The adversarial policy parameter which aims at shifting the unified policy parameter away from achieving the goal.

If the update carried out by the adversarial agent is larger than the sum of each agent's policy gradient update, the policy parameter will start moving away from the desired consensus θ^*. The success of the adversarial attack hence depends on,

- The nature of adversarial policy parameter θ_l^{k-}

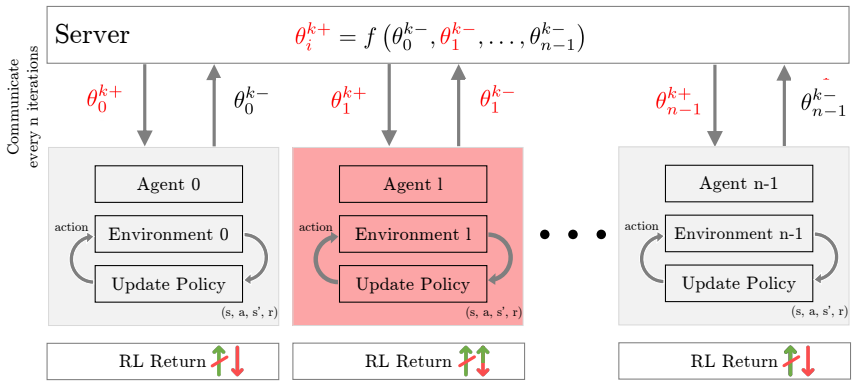


Figure 4.2: Adversaries can negatively impact the unified policy by providing adversarial policies to the server. This results in negatively impacting the achieved discounted return on the environments

- Non-adversarial agent's local learning rate δ_i
- The number of non-adversarial agents $n - |\mathcal{L}|$

An adversarial attack is more likely to be successful if the local learning rate of non-adversarial agents δ_i is small and the number of adversarial agents $|\mathcal{L}|$ is large.

Threat Model: For an adversarial agent to be successful in its attack, it needs to shift the convergence from θ^* in Equation 4.7 to θ' such that the resultant policy π' follows

$$\mathbb{E}_{s_i \sim \rho_i} V_i^{\pi'}(s_i) << \mathbb{E}_{s_i \sim \rho_i} V_i^{\pi^*}(s_i), \quad \forall i \notin \mathcal{L} \tag{4.11}$$

The only way an adversarial agent can control this convergence is through the policy parameter θ_l^{k-} that it shares with the server. The adversarial agent needs to share the policy parameter that moves the distribution of the smoothing average of non-adversarial agents either to a uniform distribution or in the direction that purposely yields bad actions (Figure 4.3). Generally shifting the distribution to uniform distribution will require less energy than shifting it to a non-optimal action distribution. This requires that the adversary cancel out the information gained by all the other non-adversarial agents hence not being able to differentiate between good and bad actions, leaving all the actions equally likely.

Threat Model: We will assume the following threat model. At iteration k, each adversarial agent l shares the following policy parameter with the server

$$\theta_l^{k-} = \lambda^k \theta_{adv}^k \tag{4.12}$$

Hence the threat model is defined by the choice of the attack model θ_{adv} and $\lambda^k \in \mathbb{R}$ which is a non-negative iteration-dependant scaling factor that will be used to control the norm of the adversarial attack. To make the scaling factor more meaningful, we will assume that

$$\|\theta_{adv}\|^2 \approx \frac{1}{(n - |\mathcal{L}|)} \sum_{i \notin \mathcal{L}} \|\theta_i\|^2$$

The relative difference in the norm of the policy parameter between the adversarial agent and non-adversarial agent will be captured in the scaling factor term λ^k. One thing we need to be mindful of is the value of the scaling factor. If the scaling factor is large enough, almost any random (non-zero) values for the adversarial policy parameter θ_l^k will result in a successful attack. We will quantify the relative performance of the threat models by

- How effective they are in the attack, either in terms of the achieved discounted return under the threat-induced unified policy or in terms of a more problem-specific goal parameter (more in the experimentation section).
- If two threat models achieve the same attacking performance, the one with a smaller scaling factor λ^k will be considered better. The smaller the norm of the adversarial policy parameter, the better the chances for the threat model to go unnoticed by the server.

4.5 Common Attack Models

In this section, we will discuss a few common attack models θ_{adv} and propose an adaptive attack model. For the rest of the section, we will focus on the single-agent adversarial

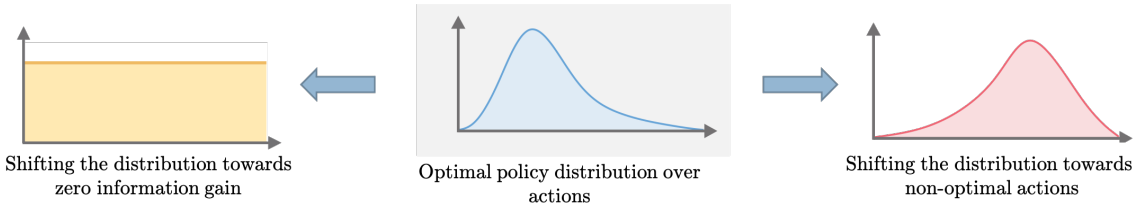


Figure 4.3: The objective of an adversarial agent is to shift the policy distribution that yields poor actions

model $|\mathcal{L}| = 1$. The extension of these threat models to multiple adversarial agents is straightforward.

4.5.1 Random Policy Attack (Rand)

This attack will be used as a baseline for the other attack methods. In a Random policy attack, the adversarial agent maintains a set of random policy parameter sampled from a Gaussian distribution with mean 0 and standard deviation $\sigma \in \mathbb{R}$ i.e. for each element $\theta_{adv,j}$ of θ_{adv}

$$\theta_{adv,j} \sim \mathcal{N}(0, \sigma) \tag{4.13}$$

This attack assumes that the adversary has no knowledge to estimate the best attack method from. If the scaling factor λ^k is large enough, this attack method can shift the distribution of the policy towards a random distribution.

4.5.2 Opposite Goal Policy Attack (OppositeGoal)

This attack method assumes that a sample environment is available for the agent to devise the attack. In this attack method, the adversary l learns a policy $\pi^{OG}_{\theta_{adv}}$ utilizing its local environment with the goal of minimizing (instead of maximizing) the long term discounted return i.e. the objective function to maximize is

$$J(\theta_{adv}) = -V_l^{\pi_{\theta_{adv}}}(\rho_l) \tag{4.14}$$

With the completion of an episode k, the adversary updates its policy parameter θ_{adv} locally by maximizing Equation 4.14 and shares the scaled version of the updated policy parameter with the server.

The OppositeGoal attack method can either shift the policy to a uniform distribution, or to a distribution that prefers actions that yield opposite goal. For the agent to shift the distribution to uniform, the following constraints need to hold.

1. All the N environments should be similar enough that they generate policies that are close enough i.e.

$$\frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} KL(\pi_{\theta_i}(.|s), \pi_{\theta_j}(.|s)) \leq \epsilon \tag{4.15}$$

   or equivalently if the learning rate and initialization is the same for each agent then

$$\left\| \frac{\theta_i}{\|\theta_i\|_2} - \frac{\theta_j}{\|\theta_j\|_2} \right\|_2 \leq \epsilon \tag{4.16}$$

2. Action selection based on minimum probability action for OppositeGoal policy should be close enough to action selection based on maximum probability for NormalGoal Policy

$$\frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} KL(1 - \pi^{OG}_{\theta_i}(.|s), \pi_{\theta_j}(.|s)) \leq \epsilon \tag{4.17}$$

   or equivalently if the learning rate is same initialization is zero $\theta_i^0 = \mathbf{0}$

$$\left\| \frac{\theta_i}{\|\theta_i\|_2} + \frac{\theta_j}{\|\theta_j\|_2} \right\|_2 \leq \epsilon \tag{4.18}$$

In short, all the N environments should be similar enough such that training on an opposite goal will yield a policy that when combined with a policy learned to maximize the actual goal will yield a complete information loss.

Most of the time, these assumptions will not hold as they are too strict (the difference in environment dynamics, initialization of policy parameter, the existence of multiple local

minima, etc.). Instead, if the scaling factor is large, the OppositeGoal attack will shift the distribution of the consensus to an opposite goal policy. Since we are taking into account the environment dynamics, this attack will however be better than the random policy attack.

4.5.3 Adversarial Attack by Minimizing Information Gain (*AdAMInG*)

Even though the adversarial choice of the opposite goal makes an intuitive sense as the best attack method, we will see in the results section that it's not. Hence, we propose an attack method that takes into account the nature of MT-FedRL smoothing averaging and devises the best attack given the information available locally. The goal of *AdAMInG* is to devise an attack that uses a single adversarial agent with a small scaling factor by forcing the server to forget what it learns from the non-adversarial agents.

For the smoothing average at the server to lose all the information gained by other non-adversarial agents we should have

$$\theta_l^{k-} = -\frac{1}{\beta^k|\mathcal{L}|}\left(\alpha^k\theta_i^{k-} + \beta^k\sum_{j\neq i,l}\theta_j^{k-}\right) \tag{4.19}$$

Using the above equation in Equation 4.8 will result $\theta_i^{k+} = \mathbf{0}$, hence losing the information gained by θ_i^{k-}. The problem in the above equation is that the adversarial agents do not have access to the policy parameter shared by non-adversarial agents $\theta_i^{k-}, \forall i \neq l$ and hence the quantity in the parenthesis (smoothing average of the non-adversarial agents) is unknown. The attack model then is to estimate the smoothing average of the non-adversarial agents.

The adversarial agent has the following information available to it

- The last set of policy parameter shared by the adversarial agent to the server $\theta_l^{(k-1)-}$
- The federated policy parameter shared by the server to the adversarial agent $\theta_l^{(k-1)+}$

The adversarial agent can estimate the smoothing average of the non-adversarial agents from these quantities. The *AdAMInG* attack shares the following policy parameter

$$\theta_l^{k-} = \lambda^k \left(\frac{\alpha^k \theta_l^{(k-1)+} - \theta_l^{(k-1)-}}{\beta^k} \right) \tag{4.20}$$

The smoothing average at the server for $i \in \{0, n-1\}, i \neq l$ becomes

$$\begin{aligned}
\theta_i^{k+} &= \alpha^k \theta_i^{k-} + \beta^k \sum_{i \neq j,l} \theta_j^{k-} + \beta^k \theta_l^{k-} \\
&= \alpha^k \theta_i^{k-} + \beta^k \sum_{i \neq j,l} \theta_j^{k-} - \\
&\quad \frac{\lambda^k}{n-1} \alpha^k \left(\theta_l^{(k-1)+} - \theta_l^{(k-1)-} \right) \\
&= \alpha^k \theta_i^{k-} + \beta^k \sum_{i \neq j,l} \theta_j^{k-} - \frac{\lambda^k}{n-1} \beta^k \sum_{j \neq l} \theta_j^{(k-1)-} \\
&= \left(\alpha^k \theta_i^{k-} - \frac{\lambda^k}{n-1} \beta^k \theta_i^{(k-1)-} \right) \\
&\qquad + \sum_{j \neq i,l} \left(\beta^k \theta_j^{k-} - \frac{\beta^k \lambda^k}{n-1} \theta_j^{(k-1)-} \right)
\end{aligned}$$

We want $\theta_i^{k+} \rightarrow \mathbf{0},\ \forall i \in \{0, n-1\}, i \neq l$. This means forcing the two terms inside the parenthesis to $\mathbf{0}$. If the initialization of all the agents are same, i.e. $\theta_i^{0-} = \theta^0 = \mathbf{0}\ \forall i$ and the learning rate is small, we have $\|\theta_i^{k-} - \theta_i^{(k-1)-}\| < \epsilon$. Hence $\theta_i^{k+} \rightarrow \mathbf{0}$ can be achieved by the following scaling factor

$$\lambda^{k*} = argmin_{\lambda^k}\ g(\lambda^k, n)$$

where

$$g(\lambda^k, n) = \left| \alpha^k - \beta^k \frac{\lambda^k}{n-1} \right| + \left| \beta^k \left(1 - \frac{\lambda^k}{n-1} \right) (n-2) \right|$$

For simplicity we have not shown the dependence of α^k, β^k in the expression $g(\lambda^k, n)$ as

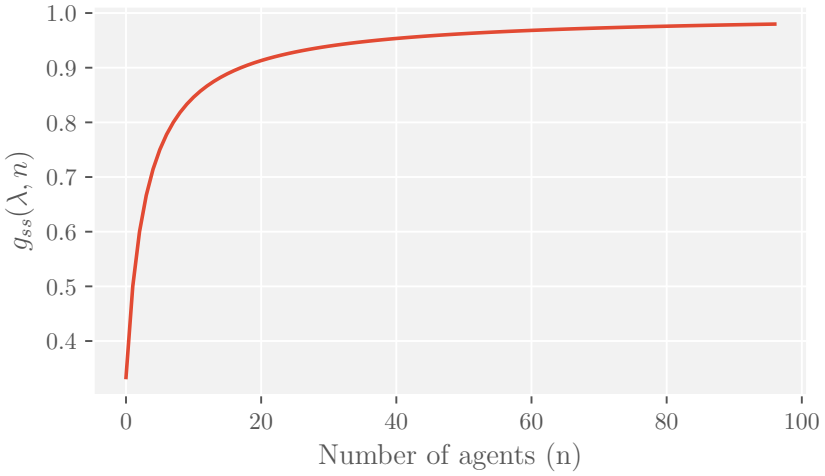


Figure 4.4: $g\left(\lambda^k, n\right)$ as a function of $\lambda^k = 1$ and n

they directly depend on k. Solving this optimization problem yields

$$\lambda^* = n - 1, \quad n \geq 3 \tag{4.21}$$

This means that the scaling factor should be equal to the number of non-adversarial agents and is independent of the iteration k. For $\lambda^k < \lambda^*$ we still can achieve a successful attack if the learning rate δ is not too high.

As the training proceeds, the values of the smoothing constants α^k, β^k approach their steady-state value of $\frac{1}{n}$. At that point, the steady-state value of $g(\lambda^k, n)$ defined as $g_{ss}(\lambda^k, n)$ is given by

$$g_{ss}(\lambda^k, n) = \frac{n - 1 - \lambda}{n} \tag{4.22}$$

The steady-state value $g_{ss}(\lambda^k, n)$ signifies how effective/successful the *AdAMInG* attack will be for the selected parameters (λ^k, n). A steady-state value of 0 signifies a perfect attack, where the policy parameter shared by the server loses all the information gained by the non-adversarial agents. On the other hand, a steady-state value of 1 indicates a completely unsuccessful attack. The smaller the $g_{ss}(\lambda^k, n)$, the better the *AdAMInG* attack.

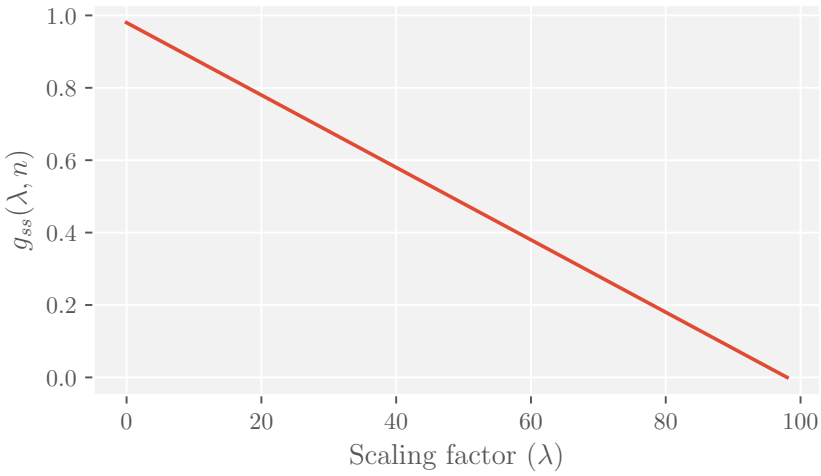


Figure 4.5: $g\left(\lambda^k, n\right)$ as a function of λ^k and $n = 100$

Figure 4.4 plots $g(\lambda^k, n)$ as a function of the number of agents n for a scaling factor of 1 ($\lambda^k = 1$). It can be seen that as the number of agents increases, the steady-state value g_{ss} becomes closer to 1 making it difficult for *AdAMInG* to have a successful attack with a scaling factor of 1. As the number of agents increases, the update carried out by the non-adversarial agent has a more significant impact on the smoothing average than the adversarial agent making it harder for the adversarial agent to attack. Figure 4.5 plots $g(\lambda^k, n)$ as a function of the scaling factor λ^k for $n = 100$. It can be seen that the scaling factor has a linear impact on the success of the *AdAMInG* attack. The performance of the *AdAMInG* attack increases linearly with the increase in the scaling factor. The best *AdAMInG* attack is achieved when $\lambda^k = n - 1$ which is consistent with Equation 4.21.

In the experimentation section, we will see that a non-zero steady-state value can still result in a successful attack for a small learning rate δ.

It is safe to assume that if we do not change the learning rate (and it is small enough), we can find the scaling factor required to achieve the same attacking performance by increasing the number of agents n. The steady-state relationship between n and λ in Equation 4.22 lets us analyze the relative attacking performances by varying the number of agents n. Let's say that for n_1 number of agents and a given learning rate that is small, we were able to achieve

a successful attack with λ_1. Now to achieve the same successful attack for n_2 number of agents we need

$$\lambda_2 = \frac{n_2(1+\lambda_1)}{n_1} - 1 \tag{4.23}$$

Unlike the OppositeGoal attack, we can guarantee that the *AdAMInG* attack will yield a successful attack if the scaling factor is equal to the number of non-adversarial agents.

We will see in the results section that the scaling factor needs no to be this high if the learning rate δ is not high. We will be able to achieve a good enough attack even if $\lambda^k < n - 1$. The only downside with the *AdAMInG* attack method is that it requires twice the amount of memory as compared to that of the OppositeGoal or Rand attack method. *AdAMInG* attack method needs to store both the adversary shared policy parameter $\theta_l^{(k-1)-}$ and the server shared policy parameter $\theta_l^{(k-1)+}$ from the previous iteration to compute the new set of policy parameters to be shared θ_l^{k-} as shown in Equation 4.20. However, as opposed to the OppositeGoal attack method, the *AdAMInG* attack method does not require learning from the data sampled from the environment saving up much on the compute cost.

4.6 Detecting attacks - ComA-FedRL

Evaluated Policy	**Environment**	**Cumulative reward**
Non-adv	Non-adv	High
Non-adv	Adv	Low (Secondary Attack)
Adv	Non-adv	Low
Adv	Adv	Low (Secondary Attack)

Table 4.1: Cross-evaluation of policies in `ComA-FedRL` in terms of cumulative return

We will see in section 4.7 that the FedRL algorithm under the presence of an adversary can severely affect the performance of the unified policy. Hence, we propose Communication

Algorithm 5: Communication Aware Federated RL

Initialization: Initialize number agents n, $\theta_i^0 \in \mathbb{R}^d$, step size δ^k,
$base\_comm \in \mathbb{R},\ comm[i] = base\_comm\ \forall i \in \{0, n-1\},\ wait\_comm \in \mathbb{R}$
for *k=1,2,3,...* **do**
  % Pre-train phase
  if $k \leq wait\_comm$ **then**
    if $k\%base\_comm = 0$ **then**
      for *each agent i* ***in parallel*** **do**
        $\theta_i^{(k+1)-} \leftarrow$ `ClientUpdate` $\left(i, \theta_i^{k+}\right)$
      end
      $r \leftarrow$ `CrossEvalPolicies` $\left(r, \theta^{(k+1)-}\right)$
    end
  end
  else
    Calculate smoothing average parameters α_k, β_k $comm \leftarrow$ `UpdateCommInt`$(r, comm)$
    for *each agent i* ***in parallel*** **do**
      if $k\%comm[i] = 0$ **then**
        $\theta_i^{(k+1)-} \leftarrow$ `ClientUpdate`$\left(i, \theta_i^{k+}\right)$
      end
    end
    $num\_active\_agents = 0$
    for *each agent i* **do**
      if $k\%comm[i] = 0$ **then**
        num_active_agents+=1

$$\theta_i^{(k+1)+} = \alpha^k \theta_i^{(k+1)-} + \beta^k \sum_{i \neq j} \theta_j^{(k+1)-}$$

        Send $\theta_i^{(k+1)+}$ back to client i
      end
    end
    if $num\_active\_agents = n$ **then**
      $r \leftarrow$ `CrossEvalPolicies` $\left(r, \theta^{(k+1)-}\right)$
    end
  end
end

Function `CrossEvalPolicies(`*r, θ*`)`**:**
  for *each agent i* ***in parallel*** **do**
    Randomly assign each agent i another agent j without replacement
    $r_j.append($`ClientEvaluate`$(i, \theta_j))$
  end
return r
Function `ClientUpdate(`*i,θ*`)`**:**
  1) Compute the gradient of the local value function $\frac{\partial V_i^{\pi_\theta}(\rho_i)}{\partial \theta_{s_i, a_i}}$ based on the local data;
  2) Update the policy parameter

$$\theta^- = \theta + \delta^k \frac{\partial V_i^{\pi_\theta}(\rho_i)}{\partial \theta_{s_i, a_i}}$$

return θ^-
Function `ClientEvaluate(`*i, θ*$_j$`)`**:**
  Evaluate the policy θ_j on agent i and return the cumulative reward ret
return ret

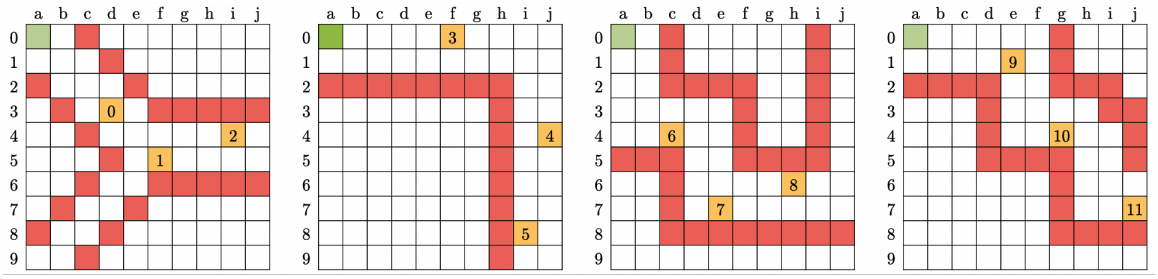


Figure 4.6: [GridWorld] The 12 environments used

Adaptive Federated RL (`ComA-FedRL`) to address the adversarial attacks on a Federated RL algorithm. Instead of communicating the policy parameter from all agents at a fixed communication interval, we assign different communication intervals to agents based on the confidence of them being an adversary. An agent, with higher confidence of being an adversary, is assigned a large communication interval and vice-versa. Communicating less frequently with an adversary agent can greatly mitigate its effects on the learned unified policy. Since we can't guarantee that a certain agent is an adversary or not, we can't just cut off the communication with an agent we think would be an adversary. Moreover, an adversary can fake being a non-adversarial agent to get away with being marked as an adversarial agent. Hence, we don't mark agents as adversary or non-adversary, rather we adaptively vary the communication interval between the server and the agents based on how good, on average, does the policy of the agent performs in other environments. The complete algorithm of `ComA-FedRL` can be seen in algorithm 5.

`ComA-FedRL` begins with a pre-train phase, where each agent tries to learn a locally optimistic policy independent of others. These locally optimistic policies are expected to perform better than a random policy on other agents' environments. After every certain number of episodes, the server randomly assigns a policy to all the environments without replacement for evaluation, and the cumulative reward achieved by this policy is recorded. Based on the nature of the policy and the environment it is cross-evaluated in, we have four cases as shown in Table 4.1. When the policy locally learned by a non-adversarial agent is evaluated in the environment of a non-adversarial agent, it generally performs better than a

random policy because of the correlation of the underlying tasks. Hence we get a slightly higher cumulative reward compared to other cases. On the other hand, if an adversarial policy is cross-evaluated on a non-adversarial agent's environment, it generally performs worse because of the inherent nature of the adversary, giving a low cumulative reward. When the policies are evaluated on the adversarial agent's environment, the adversary can present a secondary attack in faking the cumulative reward. It intentionally reports a low cumulative return with the hopes of confusing the server to mistake a non-adversarial agent with an adversarial one. Since the adversarial agent has no way of knowing if the policy shared by the server belongs to an adversarial or a non-adversarial agent, it always shares a low cumulative return.

At the end of the pre-train phase, the cumulative rewards are averaged out for a given policy and are compared to a threshold. If the averaged cumulative reward of the policy is below (above) this threshold, the policy is marked as *possibly-adversarial* (*possibly-non-adversarial*). The *possibly-adversarial* agents are assigned a higher communication interval (less frequent communication), while *possibly-non-adversarial* agents are assigned a smaller communication interval (more frequent communication). The agents are constantly re-evaluated after a certain number of iterations and the categories associated with the agents are updated. After re-evaluation, if an already marked possible-adversary agent is re-marked as *possibly-adversary*, the agent's communication interval is doubled, signifying a higher probability of it being an adversary and making it contribute even lesser towards the server smoothing average. Hence as the training proceeds, the adversarial agent's contribution to the server smoothing average becomes smaller and smaller.

Further details on the variables and functions used in algorithm 5 can be found in section A.1.

4.7 Experimentation

For the entire experimentation section, we focus on single-adversary MT-FedRL and hence $|\mathcal{L}| = 1$. We report the experimental results from a simpler tabular-based RL problem (GridWorld) to a more complex neural network-based RL problem (AutoNav). In both cases, we use policy gradient-based RL methods.

4.7.1 GridWorld - Tabular RL

Problem Description: We begin our experimentation with a simple problem of GridWorld. The environments are grid world mazes of size 10×10 as seen in Figure 4.6. Each cell in the maze is characterized into one of the following 4 categories: *hell-cell* (red), *goal-cell* (yellow), *source-cell* (green), and *free-cell* (white). The agent is initialized at the *source-cell* and is required to reach the *goal-cell* avoiding getting into the *hell-cell*. The *free-cells* in the maze can be occupied without any consequences. The agent can take one of the following 4 actions $\mathcal{A} = \{\texttt{move-up}, \texttt{move-down}, \texttt{move-right}, \texttt{move-left}\}$ which corresponds to the agent moving one cell in the respective direction. At each iteration, the agent observes a one-step SONAR-based state $s \in \mathcal{R}^4$ which corresponds to the nature of the four cells (up, down, right, left) surrounding the agent. If the corresponding cell is a *hell-cell*, *goal-cell*, or *free-cell*, the corresponding state element is -1, 1, or 0 respectively. Hence, we have $|\mathcal{S}| = 81$. Based on the nature of the environment, only a subset of these states will be available for each environment. At each iteration, the agent samples an action from the action space and based on the next state, observes a reward. The reward is -1, 1, 0.1, or -0.1 if the agent crashed into hell-cell, reached the goal, moved closer to or away from the goal respectively. The effectiveness of the MT-FedRL-achieved unified policy is quantified by the win ratio (WR) defined by

$$WR = \frac{1}{n-1} \sum_{i \neq l} \frac{\text{\# of times agent } i \text{ reached goal state}}{\text{total \# of attempts in environment } i}$$

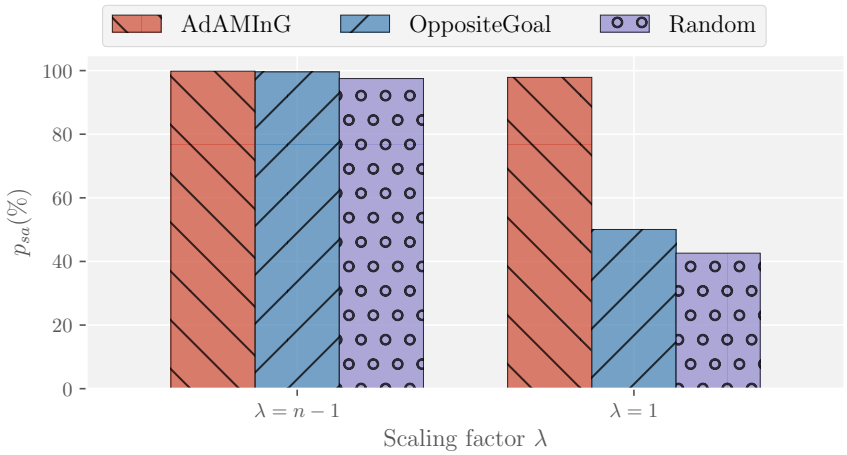


Figure 4.7: [GridWorld] Probability of successful attack $p_{sa}(\%)$ under different attack models. The greater the p_{sa} the better the performance of the adversary.

In this 12-agent MT-FedRL system, agent 0 is assigned the adversarial role ($l = 0$). The goal for agent 0 is to decrease this win ratio. We will characterize the performance of the adversarial attack by the probability of successful attack p_{sa} given by

$$p_{sa} = 1 - \frac{WR_{adv}}{WR_{no-adv}}$$

where WR_{adv} is the win ratio with an adversary, while WR_{no-adv} is the win ratio without any adversary. An attack method is said to be successful with probability p_{sa} if it is able to defeat the system $p_{sa}\%$ of the time compared to a non-adversarial MT-FedRL. The greater the p_{sa} the better the attack performance of the adversary.

Effect of Adversaries We begin the experimentation by analyzing the effect of the common attack models mentioned in section 4.5. Figure 4.7 reports the p_{sa} for the three attack methods with the scaling factor of $n - 1$ and 1 (and a learning rate $\sigma = 0.2$). With the optimal scaling factor of $n - 1$, it can be seen that all the three attack methods were able to achieve a good enough attack ($p_{sa} > 96\%$). For a scaling factor of 1, however, only *AdAMInG* attack method was able to achieve a successful attack ($p_{sa} = 98\%$). Both the random policy attack and OppositeGoal attack were only half as good as the *AdAMInG* attack method with OppositeGoal being only slightly better than a random policy attack.

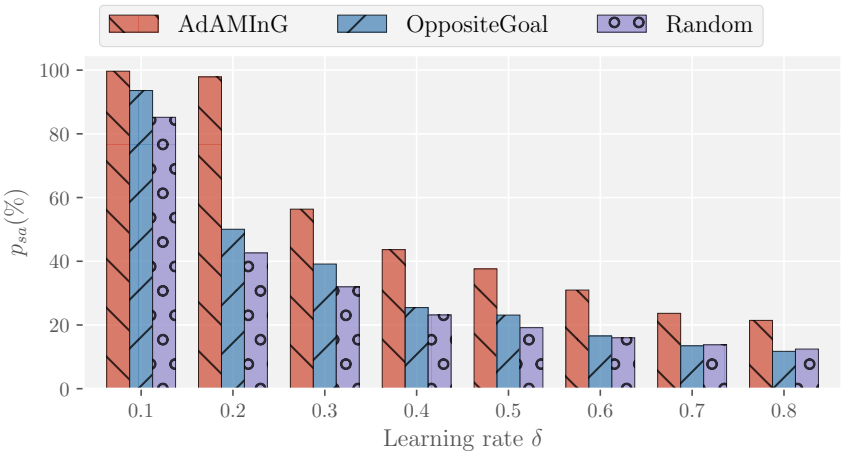


Figure 4.8: [GridWorld] Effect of learning rate (δ) on the performance of attack methods with $\lambda = 1$ and $n = 12$.

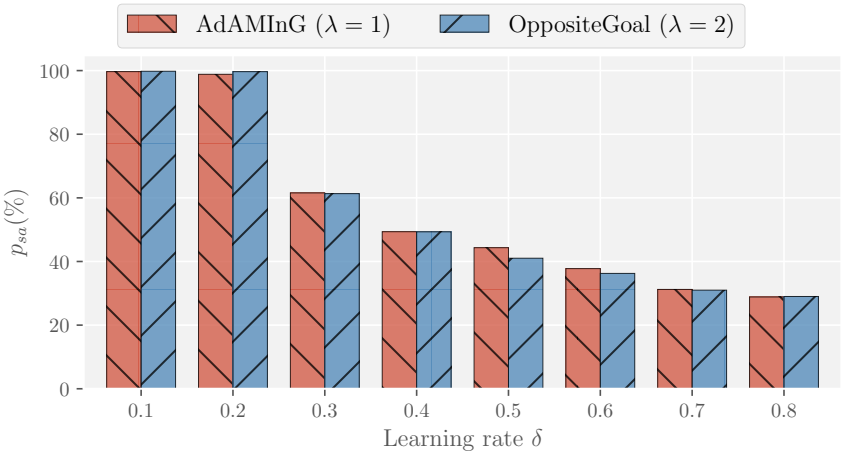


Figure 4.9: [GridWorld] Comparing attack performance for $n = 12$ between *AdAMInG* with $\lambda = 1$ and OppositeGoal with $\lambda = 2$.

As mentioned in Equation 4.4, for a scaling factor of 1, the performance of the attack method depends on the learning rate (δ) and the number of non-adversarial agents ($n - |\mathcal{L}|$).Figure 4.8 reports p_{sa} of the attack methods with varying learning rates. It can be seen that the greater the learning rate, the poorer the performance of the attack method in terms of p_{sa}. For a higher learning rate, the local update for each agent's policy parameter has more effect than the update of the server carried out with the adversary, and hence poorer the performance of the attack. Another thing to observe is that as the learning rate increases the relative performance of the OppositeGoal attack compared to the Random

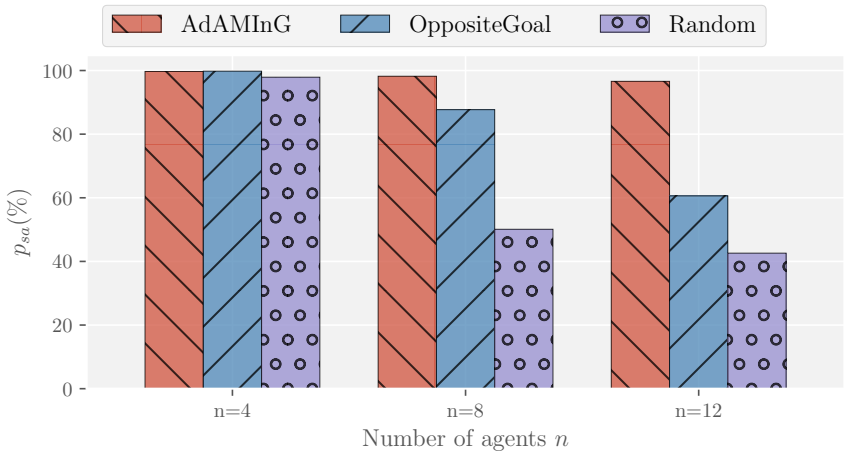


Figure 4.10: [GridWorld] Effect of number of agents (n) on the performance of attack methods with $\lambda = 1$ and $\delta = 0.2$

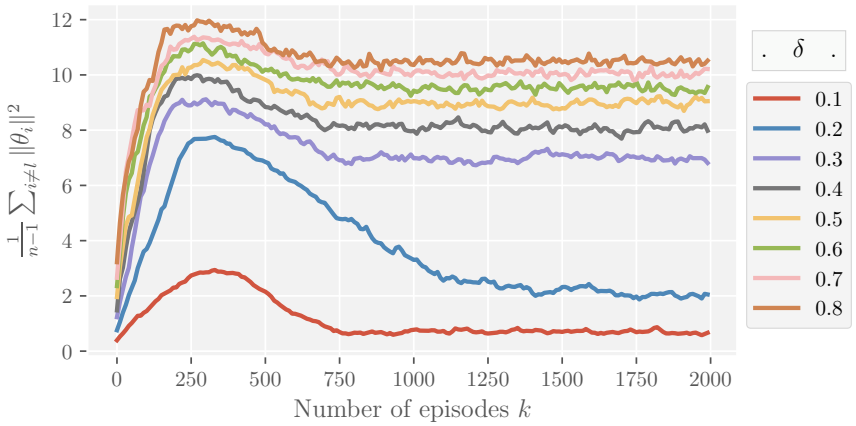


Figure 4.11: [GridWorld] Based on the learning rate, the consensus gets converged to an intermediate value

policy attack becomes poorer even becoming worse than the Random policy attack. The reason behind this is that the observable states across environments are non-overlapping. The environment available to the adversary for devising OppositeGoal attack might not have access to the states observable in other environments. Hence the OppositeGoal policy attack can not modify the policy parameter related to those states. OppositeGoal policy attack method either require a large scaling factor or more than one adversary to attack the MT-FedRL with performance similar to *AdAMInG* with single-adversary and unity scaling factor λ. This can be seen in Figure 4.9.

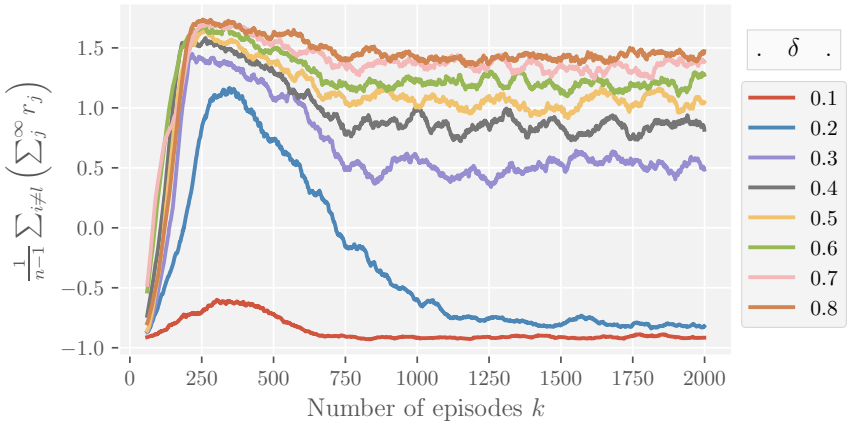


Figure 4.12: [GridWorld] Cumulative return (moving average of 60) for different learning rate (δ) and $n = 12$

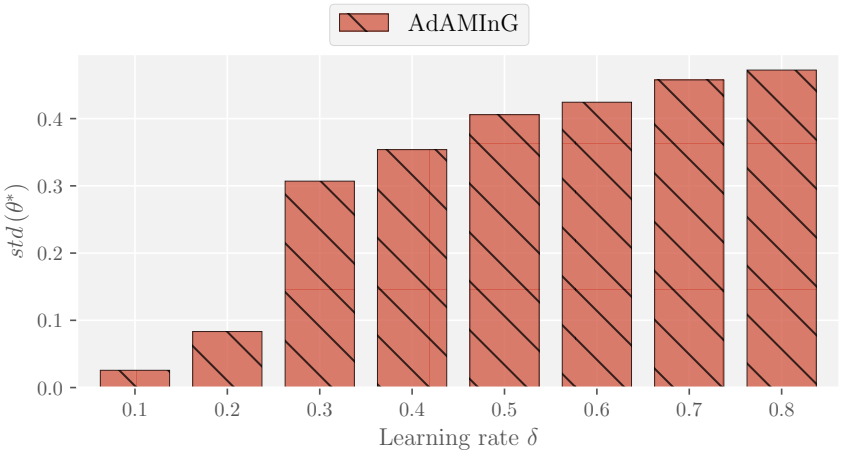


Figure 4.13: [GridWorld] Standard deviation of the consensus policy parameter

A similar trend can be observed with varying the number of non-adversarial agents. It can be seen in Figure 4.10 that for a smaller number of non-adversarial agents (equivalently smaller number of total agents if the number of the adversarial agents is fixed), it is easier for the adversary to attack with a high p_{sa}. The reason behind this is that the local update in Equation 4.10 is proportional to the number of non-adversarial agents. With a smaller number of non-adversarial agents, the local update is smaller compared to the update by the adversary. Among the three attack methods, *AdAMInG* is the most resilient to these two parameters (λ, n), hence making it a better choice for an adversarial attack in an MT-FedRL setting.

Analyzing *AdAMInG* Attack: We carry out a detailed analysis of the *AdAMInG* attack method. The smoothing average (Equation 4.10) in the presence of an adversary carries out two updates - the local update which moves the policy parameter in a direction to maximize the collective goal, and the adversarial update which tries to move the policy parameter away from the consensus. When the training begins, the initial set of policy parameters θ_i is farther away from the consensus θ^*. Gradient descent finds a direction from the current set of policy parameters to the consensus. This direction has a higher magnitude when the distance between the current policy parameter and the consensus is high. As the system learns, the current policy parameter gets closer to the consensus, and hence the magnitude of the direction of update decreases. So even if we have a static learning rate δ, the magnitude of local update $\delta_j \nabla_{\theta_j} V_j^{\pi_{\theta_j}}(\rho_j)$ in Equation 4.10 will, in general, decrease as the system successfully learns. There will be a point in training where the local update will become equal but opposite to the update being carried out by the *AdAMInG* adversary. From that point onwards, the current policy parameter won't change much. This can be seen in Figure 4.11. The greater the learning rate δ, the earlier in training we will get to the equilibrium point, and hence poorer the attack performance which can be seen in terms of the achieved discounted return in Figure 4.12. A greater standard deviation of the consensus policy parameter indicates a better differentiation between good and bad actions for a given state.Figure 4.13 plots the standard deviation of the consensus policy parameter for different learning rates δ. It can be seen that for higher learning rates, the consensus has a higher standard deviation hence being able to perform better than the consensus achieved under lower learning rates.

We also compare the performance of the *AdAMInG* attack in relation to the scaling factor λ and the number of agents n. AEquation 4.23n increase in the number of agents can be compensated by increasing the scaling factor λ to achieve the same attacking performance. We analyse the *AdAMInG* attack for the following two configurations: $(\lambda = 1, n = 8)$ and $(\lambda = 2, n = 12)$. Table 4.2 reports the p_{sa} and the standard deviation of the consensus pol-

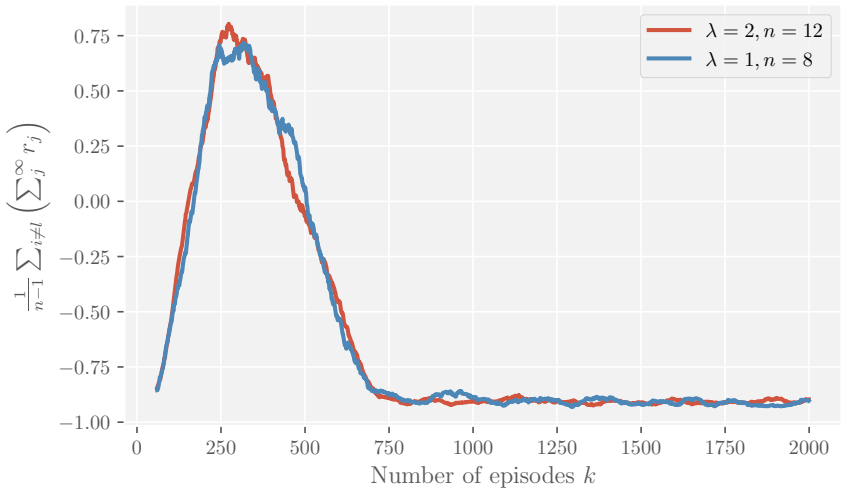


Figure 4.14: [GridWorld] Relationship between λ and n for same *AdAMInG* attack performance. $(\lambda = 1, n = 8)$ and $(\lambda = 2, n = 12)$ follows the same discounted return across episodes which is in accordance with Eq. Equation 4.23

icy parameter θ^*. It can be seen that both configurations generate similar numbers. The same trend can be observed temporally, in Figure 4.14, for the achieved discounted return during each episode in training.

Configuration	**Learning rate** δ	$p_{sa}\%$	std
$\lambda = 1, n = 8$	0.2	99.75%	0.036
$\lambda = 2, n = 12$	0.2	99.49%	0.031

Table 4.2: [GridWorld] Relationship between λ and n for same attack performance with *AdAMInG*

Resolving adversaries: We implement the N-agent single-adversary MT-FedRL problem using `ComA-FedRL` to address the high p_{sa} of the conventional FedRL algorithm.Figure 4.15 compares the performance of FedRL and `ComA-FedRL` for different attack methods. By assigning a higher communication interval to the probable adversary, `ComA-FedRL` was able to decrease the probability of successful attack p_{sa} in the presence of adversary to as low as $< 10\%$. The mean communication interval for adversarial and non-adversarial agents is plotted in Figure 4.16. It can be seen that Random policy attack has a slightly

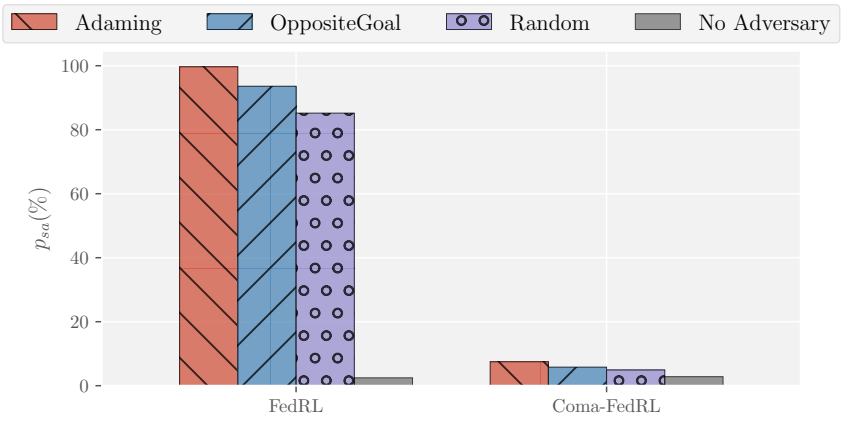


Figure 4.15: [GridWorld] Comparison of probability of successful attack $p_{sa}(\%)$ under different attack models for `FedRL` and `ComA-FedRL`. The effect of adversarial agent is greatly reduced with `ComA-FedRL`.

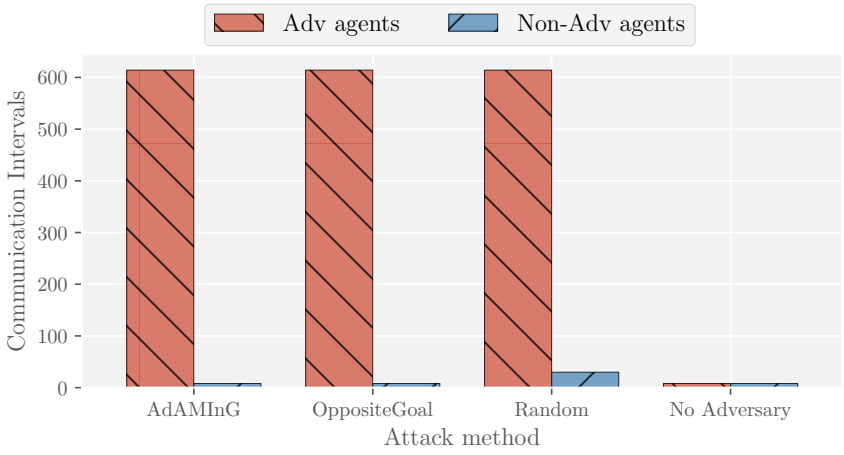


Figure 4.16: [GridWorld] Average communication intervals for adversarial and non adversarial agents in `ComA-FedRL`

higher communication interval. The reason behind this is one of the non-adversarial agents was incorrectly marked as a probable adversarial agent at the beginning of training, but later that was self-corrected to a *possibly-non-adversarial* agent.

4.7.2 AutoNav - NN based RL

Problem Description: We also experiment on a more complex problem of drone autonomous navigation in 3D realistic environments. We use PEDRA [110] as the drone navigation platform. The drone is initialized at a starting point and is required to navigate

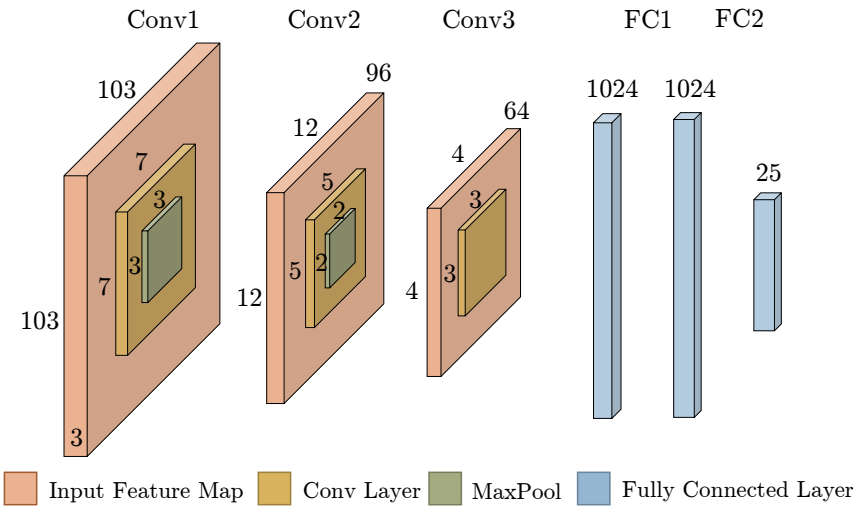


Figure 4.17: [AutoNav] C3F2 neural network used to map states to action probabilities

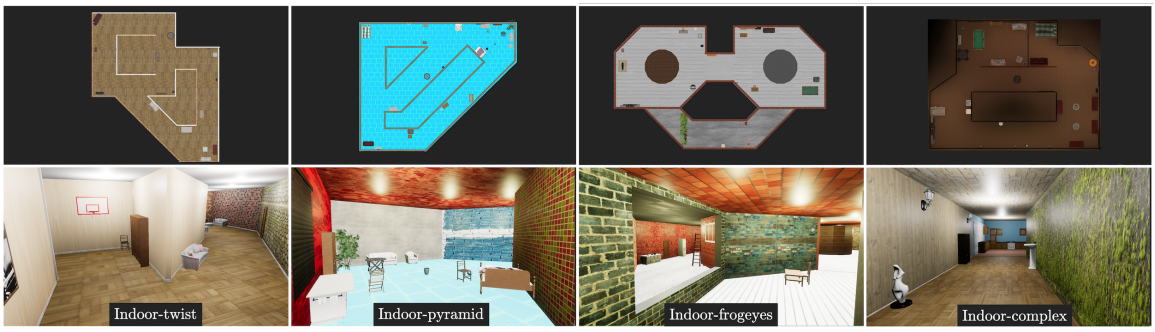


Figure 4.18: [AutoNav] Floor plan and screenshot of the four 3-D environments used

across the hallways of the environment. There is no goal position, and the drone is required to fly avoiding the obstacles as long as it can. At each iteration t, the drone captures an RGB monocular image from the front-facing camera which is taken as the state $s_t \in \mathbb{R}^{(320\times180\times3)}$ of the RL problem. Based on the state s_t, the drone takes an action $a_t \in \mathcal{A}$. We consider a perception based probabilistic action space with 25 actions ($|\mathcal{A}| = 25$). A depth-based reward function is used to encourage the drone to stay away from obstacles. We use neural network-based function approximation to estimate the action probabilities based on states. The C3F2 network used is shown in Figure 4.17. We consider 4 indoor environments (indoor-twist, indoor-frogeyes, indoor-pyramid, and indoor-complex) hence we have $n = 4$. These environments can be seen in Figure 4.18.

The effectiveness of MT-FedRL-achieved unified policy is quantified by Mean Safe

Flight (MSF) defined as

$$MSF = \frac{1}{n-1}\mathbb{E}\left[\sum_{i \neq l} d_i\right]$$

where d_i is the distance traveled by the agent in the environment i before crashing. In this 4-agent MT-FedRL system, the agent in the environment indoor-complex is assigned the adversarial role ($l = 3$). The goal for the adversarial agent is to decrease this MSF. We will characterize the performance of the adversarial attack by the probability of successful attack p_{sa} given by

$$p_{sa} = 1 - \frac{MSF_{adv}}{MSF_{no-adv}}$$

where MSF_{adv} is the mean safe flight of the MT-FedRL system in the presence of the adversary, while MSF_{no-adv} is the mean safe flight of the MT-FedRL system in the absence of the adversary. The greater the p_{sa} the better the attack method in achieving its goal.

Effect of Adversaries: For each experiment, the MT-FedRL problem is trained for 4000 episodes using the REINFORCE algorithm with a learning rate of 1e-4 and $\gamma = 0.99$. Training hyper-parameters are listed in section A.1 in detail. Table 4.3 reports the MSF achieved by the AutoNav problem for various attack methods. It can be seen that except for the *AdAMInG* attack, the rest of the attack methods achieve MSF comparable to the one achieved in the absence of an adversary ($\sim 1000m$). Figure 4.19 plots the p_{sa} for the different attack methods. It can be seen that *AdAMInG* achieves a p_{sa} of $\sim 99.5\%$ while all the other attack methods achieve a p_{sa} of $< 6\%$. The trend is similar to what was observed in the GridWorld task

Resolving Adversaries: We implement the N-agent single-adversary MT-FedRL problem using `ComA-FedRL` to address the low MSF of FedRL. The results are reported in Table 4.3. It can be seen that the decrease in MSF due to adversary was recovered using `ComA-FedRL`.Figure 4.19 plots the p_{sa} for various attack methods with `ComA-FedRL`

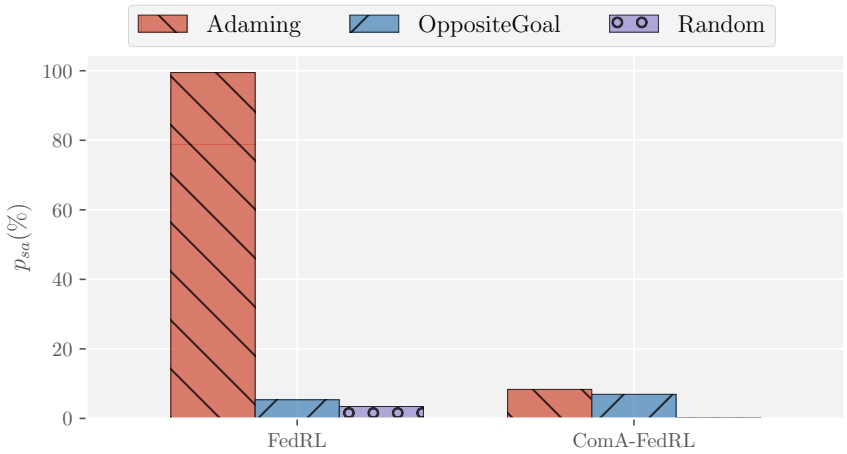


Figure 4.19: [AutoNav] Comparison of probability of successful attack $p_{sa}(\%)$ under different attack models for `FedRL` and `ComA-FedRL`. The effect of adversarial agent is greatly reduced with `ComA-FedRL`.

and compares it with `FedRL`. It can be see that with `ComA-FedRL` we have $p_{sa} < 10\%$. Hence `ComA-FedRL` was able to address the issue of adversaries in a MT-FedRL problem.

	AdAMInG	**Opposite Goal**	**Random**	**No Adv**
FedRL	6	1076	1098	1137
ComA-FedRL	1042	1028	1134	1156

Table 4.3: [AutoNav] MSF (m) for different attack methods

4.8 Summary

In this chapter, we analyze the Multi-task Federated Reinforcement Learning algorithm with an adversarial perspective. We analyze the attacking performance of some general attack methods and propose an adaptive attack method *AdAMInG* that devises an attack taking into account the aggregation operator of federated RL. The *AdAMinG* attack method is formulated and its effectiveness is studied. Furthermore, to address the issue of adversaries in the MT-FedRL problem, we propose a communication adaptive modification to conventional federated RL algorithm, `ComA-FedRL`, that varies the communication frequency

for the agents based on their probability of being an adversary. Results on the problems of GridWorld (maze solving) and AutoNav (drone autonomous navigation) show that the *AdAMInG* attack method outperforms other attack methods almost every time. Moreover, `ComA-FedRL` was able to recover from the adversarial attack resulting in near-optimal policies.

CHAPTER 5

STT MRAM BASED PROCESSING-IN-MEMORY DNN ACCELERATOR

Up until now, we explored and reported various algorithmic methods to improve the energy efficiency of an RL system. Since these algorithms will still be implemented on general-purpose compute architectures (such as CPU and GPU), we can only get enough improvements in terms of energy and latency. In order to further increase energy efficiency, we need to look at custom compute architectures or even custom compute devices.

In the next chapter, we move onto a hardware-based approach to improve the energy efficiency of the underlying RL system. We develop a novel logic embeddable STT MRAM processing-in-memory DNN accelerator which provides significant improvement in the energy efficiency for the underlying network load.

5.1 Introduction

The high compute and memory demands of the DNNs make them hard to fit in power-constrained edge devices. The severe slowdown of Moore's Law has exacerbated the burgeoning gap between the application demand and the hardware compute/memory supply. Memory-centric PIM has been proposed to accelerate machine learning applications for inference with its focus on bringing computing inside memory bitcells. This addresses the logic-to-memory bottleneck and the reduced technology improvements that currently plague general-purpose compute like CPUs and GPUs when applied to ML. However, to analyze, benchmark, and optimize a PIM-based architecture, a full-stack end-to-end simulator and optimizer is needed that can encompass different levels of hierarchies. Optimizations and innovations at all levels of this end-to-end hierarchy are needed to produce a system that can provide very high performance within an acceptable power budget. In this chapter we outline such a hierarchical and modular simulator that is used 1. to evaluate

system impacts of two novel concepts at two different layers of hierarchy, a novel logic embeddable 2T2MTJ bitcell and an ultra-pipelined mapping scheme 2. to modularly analyze critical parameters across the stack for highest power-performance and finally 3. to compare the PIM system to a digital custom ASIC framework and quantify improvements in power-performance layer-by-layer on widely accepted MLPerf benchmarks.

5.2 High Ion/Ioff 2T2MTJ bitcell and PIM Array

PIMs accelerate the most ubiquitous mathematical operation in machine learning applications, the matrix-vector multiplication. Many novel memories, e.g. resistive RAM (RRAM) and phase-change memory (PCM) have been proposed as the bitcell solutions due to their non-volatile, analog nature. However, none of these memories have been shown to be embeddable in a logic manufacturing process (both in terms of the process as well as the operating voltages), which is critical to building a machine learning processor with its requirement of myriad high-performance digital parts. Magnetic Tunnel Junction (MTJ) based Spin Transfer Torque (STT) Magnetic RAM (MRAM) bitcell is a logic embeddable non-volatile memory [111] that has hitherto been ignored for PIM applications due to its binary storage and low Ion/Ioff ratio. To overcome the digital nature of MTJs, bitsliced digital voltage signals are used to represent the input and output feature maps. Multiple input activations are fed and weighted by the bitcell conductances thereby performing MVMs simultaneously in parallel. Resultant currents on the bitline are accumulated as partial sums for an entire subarray in one clock cycle. To add the generated currents from multiple MVMs in parallel, the requirements for the bitcells are stringent. The bitcells require a high Ion/Ioff ratio as well as very low variation in conductances. RRAM and PCM memories have modestly high Ion/Ioff ratios but very high conductance variability [112]. STT MRAM is embeddable in a logic process and exhibits low conductance variation but also very low Ion/Ioff ratios. To solve the problem of the low Ion/Ioff ratio, we propose a novel cross-coupled 2T2MTJ STT MRAM bitcell in which the Ioff is determined by the

leakage of the transistor rather than the low conductance level of the MTJ, leading to on/off ratios of $> 10^4$ instead of < 3. The 2T-2MTJ bitcell is shown in Fig 1. A logical 1(0) is implemented by setting the MTJ to high resistance state Rap(Rp) and $\overline{MTJ}$ to the low resistance state Rp(Rap). The cross-coupled 2T-2MTJ bitcell enables significant advantages when the inputs are a logical 1 due to current to BL being limited by the off-state resistance of the transistor, which is typically several orders of magnitude larger than Rap as shown in Fig 2. Therefore, an array of 2T-2MTJ bitcells generates an output current on the BL that is significantly smaller than the equivalent 1T-1MTJ array enabling significant improvements in area, power, and performance as compared to a 1T-1MTJ solution [113]. The write scheme for an array comprised of 2T-2MTJ bit cells is described in Figure 5.2. The 2T-2MTJ structure is indeed 2x in the area at the bitcell level. However, the total array size (bitcell + periphery) is reduced as compared to the 1T1M implementation. This counter-intuitive fact transpires due to a significantly smaller MUX in the periphery of the 2T-2MTJ structure. The 3 to 4 orders of magnitude lower currents produced by the 2T-2MTJ structure Figure 5.1reduces the MUX-dominated total area significantlyFigure 5.3. A more detailed version of the comparison w.r.t. 1T-1MTJ and other structures has been submitted for publication at ISCAS 2020 [113]. [114] describes a bitcell that is for TCAM arrays. However, the difference with our proposal is that the reference bitcell has the MTJ resistors connected to the drain of the transistors only. The key to our proposal is the cross-coupling at the gate which leads to the 3 to 4 orders of magnitude current reduction as the off state is governed by the transistor instead of the very high MTJ R_{off}. The peripheral circuitry especially the analog-to-digital (ADC) converter plays a big role in PIM-based architectures. The ADC consumes much of the area and power and depends on the ADC bit precision which in turn affects the inference accuracy of the problem at hand eg CIFAR or ImageNet. Our modular optimizer can be used to choose the bit precision of ADCs to explore the PPA design space. As a baseline case, we use 6-bit ADC precision in conjunction with hardware-aware retraining, which has been shown to be sufficient for most

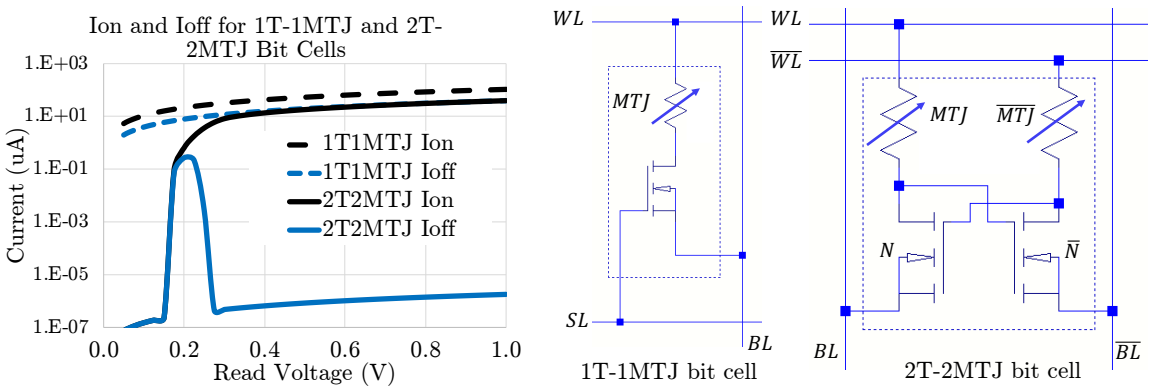


Figure 5.1: (left) Transient I_{on} and I_{off} for 1T1MTJ and 2T2MTJ bitcells clearly shows the On/Off Ratio improvement as a function of V_{read}. (right) Bitcell diagram

			Selected Column		Other Columns	
Write Up	WL	$\overline{WL}$	BL	$\overline{BL}$	BL	$\overline{BL}$
MTJ	0	V_w	V_w	V_w	0	0
$\overline{MTJ}$	V_w	0	V_w	V_w	0	0

	Selected Row		Other Rows			
Write Down	WL	$\overline{WL}$	WL	$\overline{WL}$	BL	$\overline{BL}$
MTJ	V_w	V_w	0	0	0	V_w
$\overline{MTJ}$	V_w	V_w	0	0	V_w	0

Figure 5.2: Write scheme for the 2T-2MTJ bit cell. The direction of the write (up or down) corresponds to whether the current is running from the word lines to the bit lines (down) or from the bit lines to the word lines (up). The write must take place in two separate steps – row-by-row and column-by-column

demanding ML problems [115]. The energy and area of the bitcell and peripherals are tabulated in Figure 5.3 based on 32nm node [116, 117] for consistent benchmarking wrt ScaleSim. Both negative and positive weights are encoded in our approach with multiple columns, typically 8, sharing one ADC.

5.3 XbarOpt schematic and mapping

The schematic block diagram of the XbarOpt accelerator can be seen in Figure 5.4. eMRAM is used to store the layer outputs before they can be fed into the next layer as inputs for processing. The input register is responsible for storing intermediate input activations

Area (mm2)		Energy (pJ)	
array:	0.00010485	readDynamicEnergyArray:	14.1138
wlSwitchMatrix:	0.00265971	wlSwitchMatrix:	209.938
mux:	2.33E-03	mux:	28.9239
muxDecoder:	3.26E-05	muxDecoder:	44.7752
blSwitchMatrix:	2.50E-04	multilevelSenseAmp:	140.412
multilevelSenseAmp:	0.0035144	multilevelSAEncoder:	45.4243
multilevelSAEncoder:	0.0046242	Leakege	6.54E-05
Total	**0.014517**	**Total**	**438.587**

Figure 5.3: Energy and area breakdown for bitcell array and peripherals

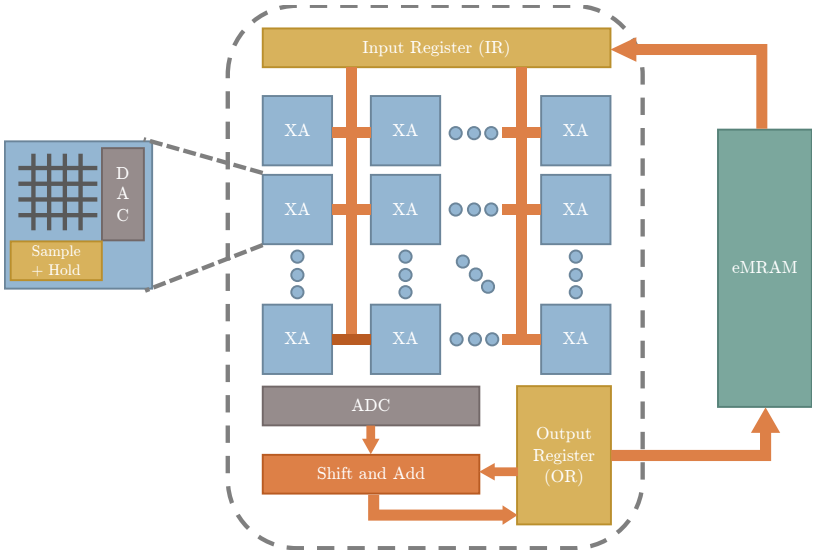


Figure 5.4: XbarOpt Schematic Diagram

to be fed into the Xbar arrays as inputs. Each xbar array unit (XA) has a DAC and Sample and Hold the unit in it. ADC is used the convert the analog outputs from the xbar arrays and the partial sums for each bit-sliced input activation are shifted and added and stored in the Output registers (OR). The idea behind XbarOpt is to allocate resources when it comes to Xbar arrays, hence it can be seen that xbar arrays are clustered together as opposed to tile-based division used in [24]. Ongoing work includes modeling these tile-based clusters into XbarOpt. A set of xbar arrays are assigned to each layer in the DNN topology. The number of rows is determined by the filter size of the layer, while the number of columns is determined by the number of filters of the layer, the number of bits mapped per cell of the STT MRAM, and the precision of the filter weights. The input activation is fed as rows to these

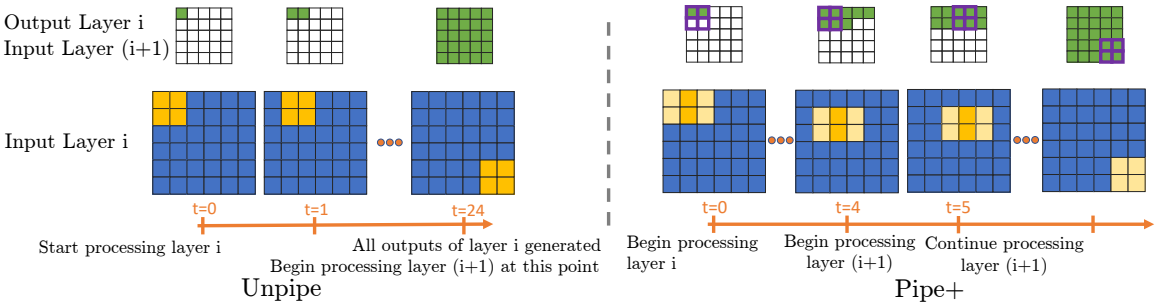


Figure 5.5: Intra layer pipelining (pipe+)

xbar arrays and the output is collected from the bottom computing the partial sum, which is then shifted and added to the partial sum of the next input activation bit. The partial sum of each output value is stored in an Output Buffer. Once the final output is generated it is stored back in the eMRAM, which can then be used as inputs to the next layer. Based on how and when this output data is used for processing in the next layer, different pipelining schemes exist. In an unpipelined (unpipe) scheme, the processing of the next layer isn't started until the entire output of the previous layer has been generated. The purpose of introducing this scheme is to have a fair comparison with a digitally implemented accelerator in the results section. In the interlayer pipelined scheme (Pipe), the processing of the next layer begins as soon as we have enough output values generated from the previous layer to [24]. This depends on the filter size and the stride length of the next layer. This scheme, however, is prone to delays when the stride length of the previous layer is greater than 1. This introduces delays in the pipeline, which trickles down to the last layer. This delay can be overcome by making use of intra-layer pipelining generating more than one output value per cycle (Pipe+ scheme). For example, if the stride length of the next layer is 2, the current layer needs to generate 2 outputs per time instant to overcome the pipeline delays caused by the lack of sufficient data. This means that we need to double the xbar arrays assigned to the current layer, which results in an increased requirement of the xbar array (and hence the requirement of a larger compute unit area). This increase in the number of xbar arrays depends on the network topology and generally is only a fraction of the total number of xbar arrays assigned to the network. Figure 5.5 visualizes these three pipelining

clk	status	address				
0	miss	0	1	2	3	4
16	hit	12	13	14	15	16
32	hit	24	25	26	27	28
48	hit	36	37	38	39	40
64	hit	48	49	50	51	52
80	hit	60	61	62	63	64
96	hit	72	73	74	75	76
112	hit	84	85	86	87	88
128	hit	96	97	98	99	100
144	hit	108	109	110	111	112
160	hit	120	121	122	123	124

clk	status	num_writes	address		
2	busy	96	20000000	20000001	20000002
3	busy	96	20000000	20000001	20000002
4	busy	96	20000000	20000001	20000002
5	busy	96	20000000	20000001	20000002
6	busy	96	20000000	20000001	20000002
7	busy	96	20000000	20000001	20000002
8	busy	96	20000000	20000001	20000002
9	done	96	20000000	20000001	20000002
10	busy	96	20000096	20000097	20000098
11	busy	96	20000096	20000097	20000098
12	busy	96	20000096	20000097	20000098

Figure 5.6: Example CSV trace file generated by XbarOpt

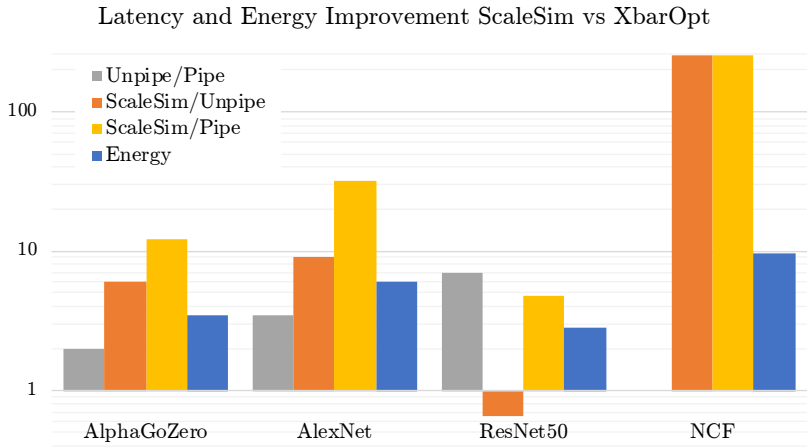


Figure 5.7: Average improvement in using XbarOpt over ScaleSim for different performance metrics (log scale)

schemes.

5.4 Results

A python-based simulator of the proposed system architecture is designed and used in conjunction with hardware-generated numbers. Prior work in the xbar arrays [118, 119, 120, 121, 122] lack a complete system architecture design and characterization of CNNs. The idea behind this simulation is to provide a full-stack understanding of the 2T2MTJ crossbar accelerator by exploring the design space for the required performance metrics. This simulation takes in the network topology, crossbar configuration file, and mapping file while outputting layer-wise and system-level performance metrics. The simulation goes through different phases and in each phase generates one or more comma-separated (CSV) files. Sample CSV trace files generated by XbarOpt can be seen in Figure 5.6. The design space

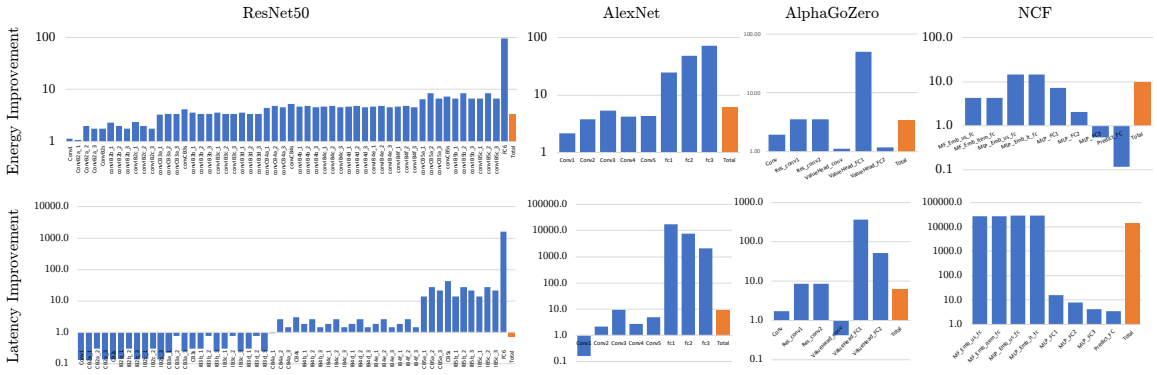


Figure 5.8: Layer-wise energy and latency improvement across all four workloads (log scale)

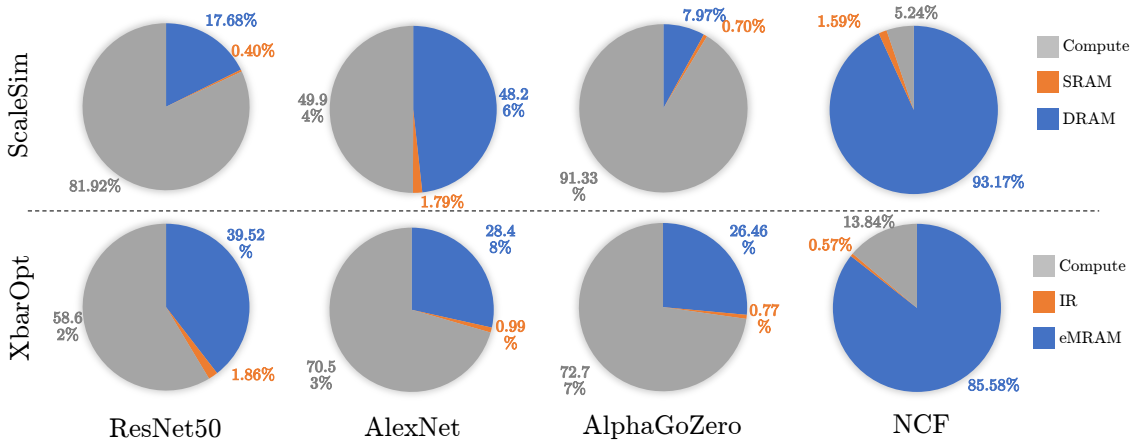


Figure 5.9: Energy breakdown for different workloads on ScaleSim and XbarOpt

includes various modeling variables such as crossbar array dimensions, 2T2MTJ modeling parameters (such as bits-per-cell, area, etc), memory sizes, access bandwidths, a selection from the various available logical-to-physical array mappings, activation precision bits, digital-to-analog (DAC) resolution of the input, analog-to-digital (ADC) precision, inter-layer-pipelining, etc.

5.4.1 XbarOpt vs Digitally-implemented Accelerator

We compare the performance of XbarOpt with a digitally implemented accelerator simulator. A systolic array accelerator based on ScaleSim [26] was used to draw the comparison which is more optimized than CPU and GPU for these MLPerf tasks. In order to have

Table 5.1: Performance parameters improvement for XbarOpt w.r.t ScaleSim

Network Name	Energy	Latency	Power	GOPS/Watt
AlphaGoZero	3.4231	12.249	0.346	2.88
AlexNet	6.1197	31.720	0.242	4.12
ResNet50	2.8628	4.8	1.379	0.72
NCF	9.6595	255.948	0.037	26.4

a fair comparison, the dimension of the systolic array used for each of the networks was determined individually such that the area occupied by the compute units for XbarOpt and ScaleSim was the same. Four different neural networks (AlphaGoZero, AlexNet, ResNet50, and Neural Collaborative Filtering) were used as workloads to both these accelerator simulators. ScaleSim is compared to both the unpipe and pipe versions of XbarOpt. Figure 5.7 summarizes the improvement results achieved by XbarOpt over ScaleSim for the following improvement performance metrics:

- Total energy per inference
- Latency between $ScaleSim$ and $XbarOpt_{unpipe}$
- Latency between $ScaleSim$ and $XbarOpt_{pipe}$
- Latency between $XbarOpt_{pipe}$ and $XbarOpt_{unpipe}$

The more the number of layers in a network, the greater the advantage of pipelining. Moreover, since there is no data re-use for fully connected layers as there is for convolutional layers, fully connected layers can't be pipelined in the manner discussed above. It can be seen from the figure that the latency improvement between $XbarOpt_{pipe}$ and $XbarOpt_{unpipe}$ is maximum for ResNet50, while it is minimum (=1) for NCF which consists of all fully connected layers and hence no room for pipelining improvement. The larger the percentage of fully connected layers (or more filters, in general) in the network, the better the resource allocation. Hence the latency improvement between $ScaleSim$ and $XbarOpt_{unpipe}$ is largest for NCF, and smallest for ResNet50. The improvement between

the latency of $ScaleSim$ and $XbarOpt_{pipe}$ depends both on the number of layers in the network and the ratio of fully connected layer weights to the total weights of the network. Hence it is the largest for NCF. The energy improvement depends on various factors, the most significant one being the xbar array utilization. Using a larger xbar array for a smaller layer will result in unnecessary energy loss. AlexNet has the best xbar array utilization and hence the best energy improvement. Table 5.1 reports the improvement in per inference energy, time, consumed power and required GOPS per watt for the workloads compared to ScaleSim.

5.4.2 Exploring the design space

In this section, we present the result of varying XbarOpt parameters and analyzing the effects on the AlexNet workload. These parameters include Xbar Array dimensions, eMRAM read/write bandwidth, pipeline techniques, ADC bit precision, and 2T2MTJ vs ReRAM based xbar arrays

The results have been plotted in Figure 5.10. Increasing the Xbar array dimensions, in general, helps in reducing the total inference energy for medium-sized filters. For workloads where the filter size is small, using larger xbar arrays will yield poor array utilization and unnecessary energy loss. Since AlexNet has a comparatively larger filter size, increasing the xbar array dimensions yields lower total inference energy. eMRAM modeling has been taken into account in terms of its bandwidth and write cycles. We observed that increasing the eMRAM bandwidth overcomes the eMRAM access delays and results in lower latency. The total energy per inference, however, depends on the per access energy of eMRAM for a given bandwidth. CACTI-[123] based numbers were used to generate the per access read and write energies. It can be seen that the energy advantage by increasing the eMRAM BW tapers off after a certain point. eMRAM suffers from low write latency. XbarOpt models the write latency of eMRAM by dictating the number of cycles required to write. For lower write cycles, the increase in the total latency per inference

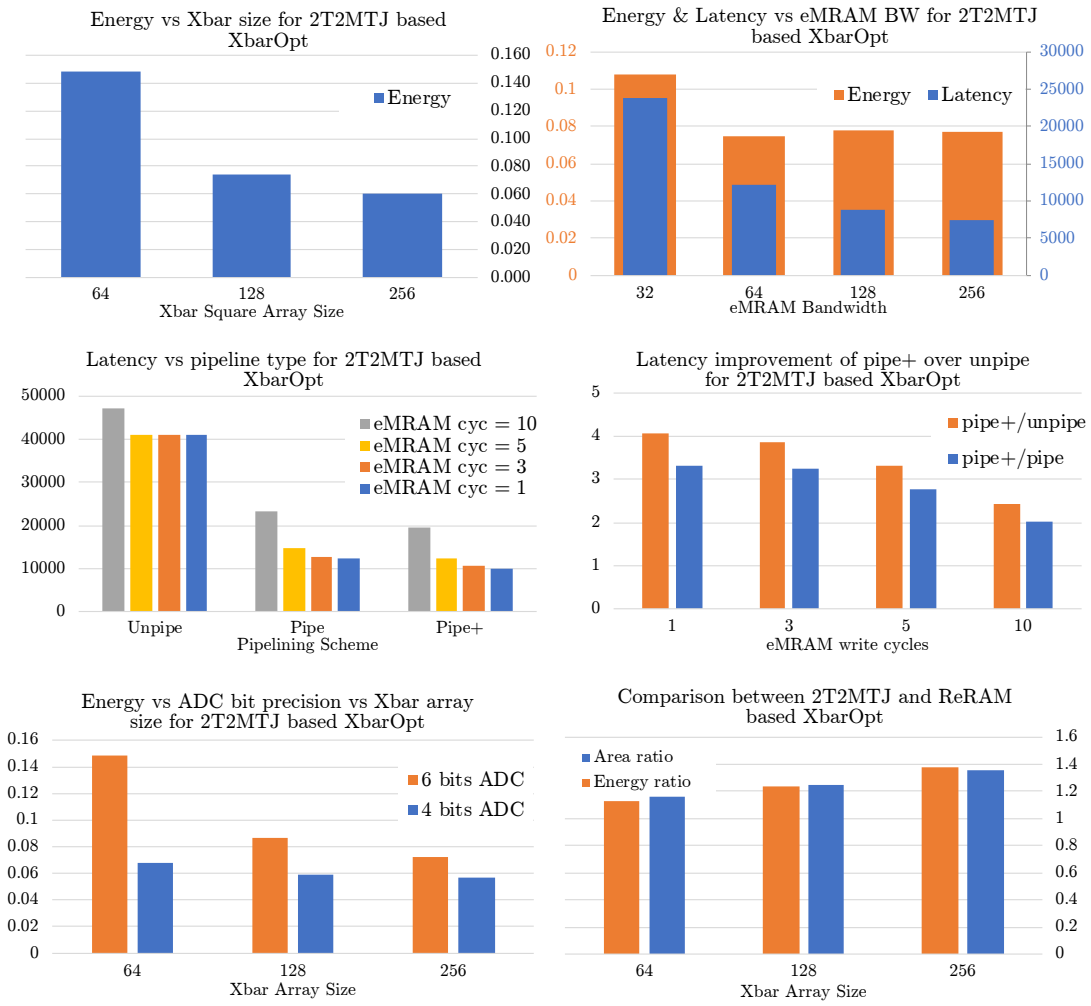


Figure 5.10: Parameter sweep over AlexNet workload

is similar. The latency increases when the write cycles increase beyond a certain value. This value is determined by the input bit slicing and precision. In these results, the input was decomposed into 8 slices of 1 bit each. Hence it takes 8 cycles for the xbar array to generate one partial sum. During that time eMRAM is not invoked and can consume cycles to write the previous partial sum. As soon as the eMRAM write cycles increases beyond 8, the total inference latency starts increasing (Figure 5.10). The total inference latency for these three pipelining schemes w.r.t to the eMRAM write cycles can be seen in Figure 5.10. The figure also shows the latency improvement of the pipe+ scheme over the unpipelined scheme as the eMRAM write cycles are varied. The pipe+ scheme yields

about 2-4 times less latency as compared to other pipelining schemes with only an extra compute area overhead of 4%. With the increase of the eMRAM write cycles, the latency improvement decreases since pipe+ is already tightly coupled and takes full advantage of parallelism, increasing the number of write cycles has a greater impact on the latency of the pipe+ scheme than that of the unpipe scheme. The last row of the figure plots the results of varying the ADC bit precision and compares the performance of the proposed 2T2MTJ Xbar design with a ReRAM based design. Increasing the ADC bit precision has a larger impact for smaller xbar arrays as compared to larger ones. For smaller xbar array sizes, the ADC dominates the total energy of the xbar array and hence a larger impact. Comparing 2T2MTJ based xbar arrays with that of ReRAM, 2T2MTJ based xbar array only yields a 1.1 to 1.4 times inference energy as that of an ideal ReRAM based xbar array design, with an added advantage that it can be fabricated.

5.5 Summary

Optimizations and innovations at all levels of hierarchy are needed to produce a PIM-based system that can provide very high performance within an acceptable power budget especially at the edge. We outline such a hierarchical and modular simulator that is used to

1. Evaluate system impacts of two novel concepts at two different layers of hierarchy, a novel logic embeddable 2T2MTJ bitcell and an ultra-pipelined mapping scheme

2. Modularly analyze critical parameters across the stack for highest power-performance and finally 3. to compare the PIM system to a digital custom ASIC framework and quantify improvements in power-performance.

CHAPTER 6

PROGRAMMABLE ENGINE FOR DRONE RL APPLICATIONS

In this chapter, we present an open-source tool PEDRA that was developed as a part of this research. The main objective of PEDRA is to provide a benchmark to test ML algorithms for drone-oriented applications in a suite of 3D realistic environments.

6.1 What is PEDRA?

PEDRA is a programmable engine for Drone Reinforcement Learning (RL) applications. The engine is developed in Python and is module-wise programmable. PEDRA is targeted mainly at goal-oriented RL problems for drones, but can also be extended to other problems such as SLAM, etc. The engine interfaces with the Unreal gaming engine using AirSim to create the complete platform. Figure 6.1 shows the complete block diagram of the engine.

Unreal Engine is used to create 3D realistic environments for the drones to be trained in. Different levels of detail are added to make the environment look as realistic or as required as possible. PEDRA comes equipped with a list of 3D realistic environments that can be selected by the user. Once the environment is selected, it is interfaced with PEDRA using AirSim. AirSim is an open-source plugin developed by Microsoft that interfaces Unreal Engine with Python [68]. It provides basic python functionalities controlling the sensory inputs and control signals of the drone. PEDRA is built onto the low-level python modules

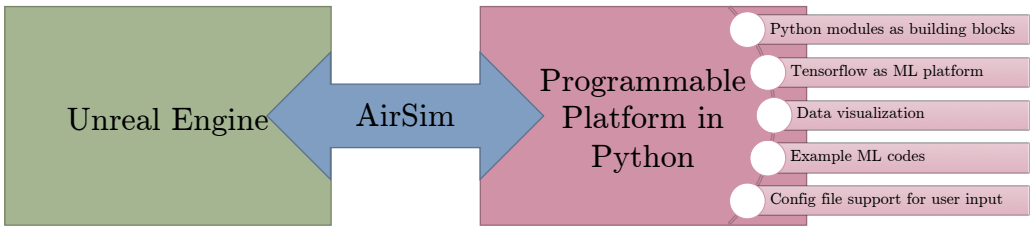


Figure 6.1: Python based training framework for drone related applications

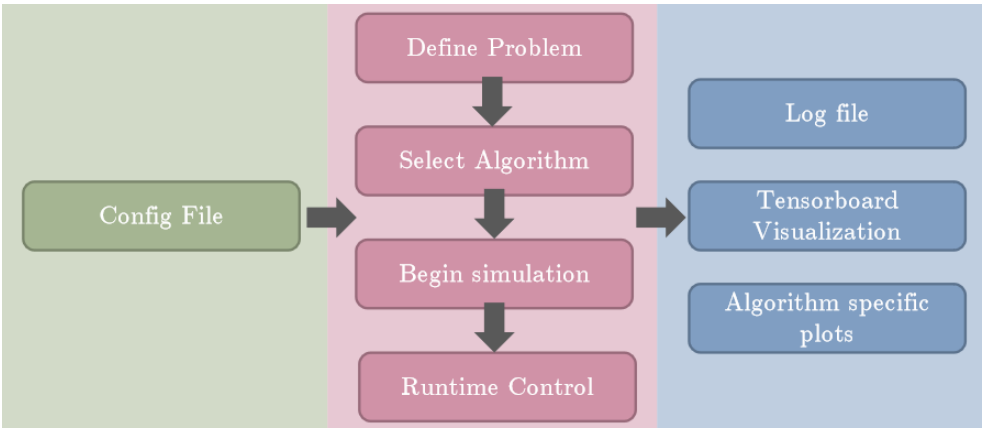


Figure 6.2: Workflow of PEDRA

provided by AirSim creating higher-level python modules for the purpose of drone RL applications.

6.2 PEDRA Workflow

The complete workflow of PEDRA can be seen in Figure 6.2. The engine takes input from a config file (.cfg). This config file is used to define the problem and the algorithm for solving it. It is algorithmic specific and is used to define algorithm-related parameters. Right now the supported problem is camera-based autonomous navigation and the supported algorithms are single/multiple drone vanilla RL, single/multiple drone PER/DDQN based RL, single/multi drone REINFORCE method for RL. More problems and associated algorithms are being added. The most important feature of PEDRA is the high-level python modules that can be used as building blocks to implement multiple algorithms for drone-oriented applications. The user can either select from the above-mentioned algorithms or can create their own using these building blocks. In case the user wants to define their own problem and associated algorithm, these building blocks can be used. Once these requirements are set, the simulation can begin. PyGame screen can be used to control stimulation parameters such as pausing the simulation, modifying algorithmic or training parameters, overwrite the config file and save the current state of the simulation, etc. PEDRA generates

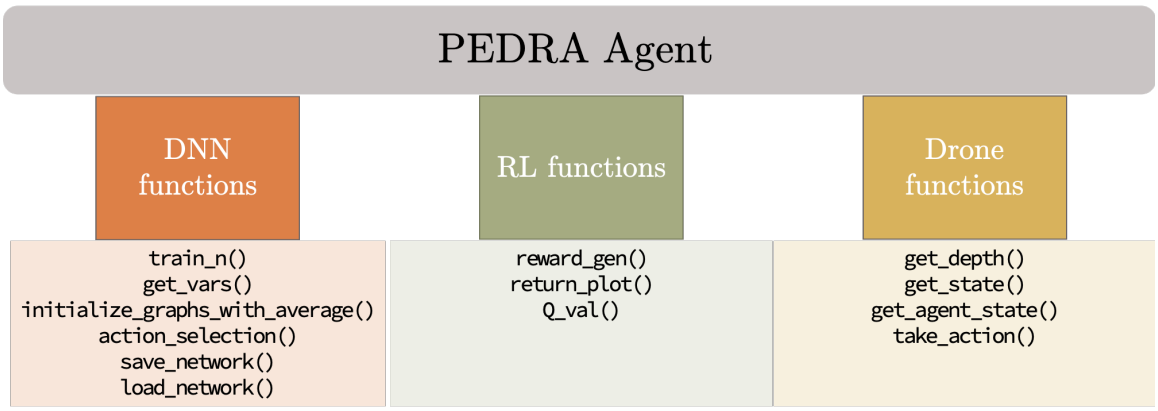


Figure 6.3: PEDRA Agent is a combination of the network model, drone, and reinforcement learning functions

a number of output files. The log file keeps track of the simulation state per iteration listing useful algorithmic parameters. This is particularly useful when troubleshooting the simulation. Tensorboard can be used to visualize the training plots in run-time. These plots are particularly useful to monitor training parameters and to change the input parameters using the PyGame screen if need be.

Each drone object is characterized by a PEDRA agent. A PEDRA agent is a modular class that contains all the necessary features, parameters, and characters required to define and control a drone agent. It is a combination of the network model, drone, and reinforcement learning functions. Users can modify these functions (or add new ones) according to their requirements if need be. PEDRA agents can also be used to define multiple agents in a distributive system.

6.3 Environments and Drone Agents

The most important part of PEDRA is the set of 3D environments. PEDRA comes equipped with a library of 3D realistic environments that can be used for drone applications. The environments fall into two categories: Indoor and Outdoor. Currently, PEDRA has 10 indoor and 3 outdoor environments that can be used for drone applications. More environments are constantly being added. These environments cover a wide range of structures, object

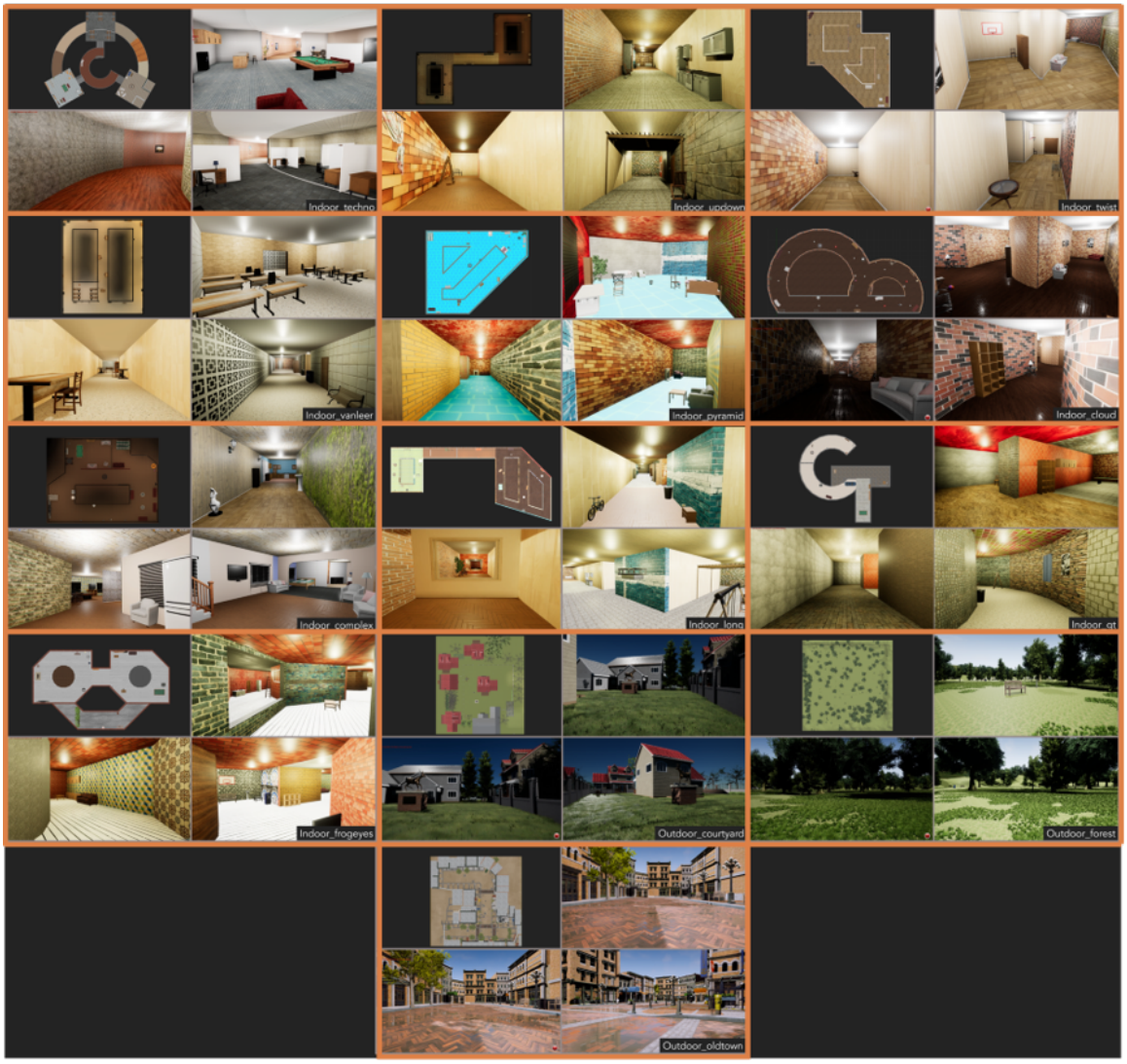


Figure 6.4: Set of available 3D realistic Indoor and Outdoor environments

nature, sizes, lighting conditions, hallways, background scenes, etc. User-generated environments can also be used with PEDRA. A screenshot of these environments with their floor plans can be seen in Figure 6.4

PEDRA has a list of physical drone agents to select from. These physical drones can be seen in Figure 6.5. These drones differ in size and shape and can be assigned different action spaces. These drones can be selected through the PEDRA config files. Different action space or flight models can be assigned to these drones. The idea behind different drones is to capture the wide spectrum of drones available in real life.

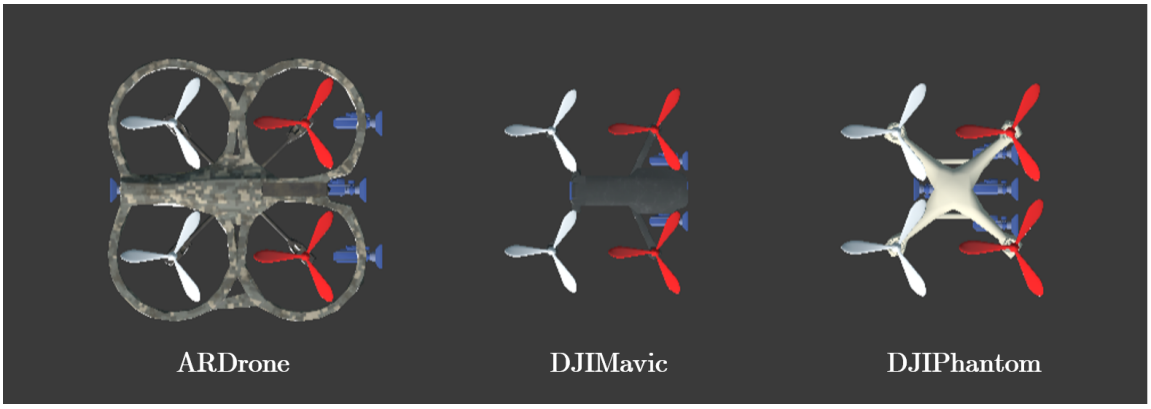


Figure 6.5: Available drone agents

6.4 Detailed Documentation and Download

PEDRA is maintained as an open-source GitHub repository. Detailed documentation on it's use, working example, and various features can be found at

https://github.com/aqeelanwar/PEDRA

CHAPTER 7
CONCLUSION

In this dissertation, the research on energy-efficient RL systems to enable edge intelligence in resource-constrained autonomous systems is presented. Among the many applications, we focused on the application of drone autonomous navigation in environments using Deep reinforcement learning. In chapter 1, we talk about the motivation of the research and discuss different prior work in this area highlighting their shortcomings. We look at various algorithmic, hardware, and co-design approaches to energy-efficient ML systems. We briefly discuss the problem of drone autonomous navigation and provide a background on reinforcement learning.

In chapter 2, we try to set up a real-world problem of drone autonomous navigation in real environments with a real drone. We highlight the key challenges of implementing reinforcement learning for drone navigation in the real environment such as reward generation, safety issues, and large online data set requirements. We address these issues by modifying the vanilla RL algorithm by incorporating the concepts of a virtual crash, data augmentation, and expert data inclusion. Experimentation on a Parrot AR Drone 2.0 showed that the proposed approach was able to outperform other baselines in terms of the drone distance traveled by overcoming the challenges mentioned above.

Chapter 3 addresses the issue of energy efficiency in such an application of drone autonomous navigation. A transfer learning-based approach to address the high energy requirement of training deep neural networks with RL was proposed. A set of 3D realistic environments were designed using Unreal Engine with various features, lighting conditions, wall colors, and objects. The network was trained for autonomous navigation in these environments collectively. After a careful analysis of the networks trained on a set of simulated environments and trained on real environments (from chapter 1), it was ob-

served that the learned weights of the initial layers of the networks were almost similar, while the weights of the end layers were different. The similarity in the distribution of the weight of earlier layers arises from the fact that those layers are responsible for capturing the high-level features of the underlying problem which was the same across the two problems. Hence it was proposed that instead of training the deep network on the real environment from scratch, using the learned weights from the network trained in simulated environments can result in significant energy efficiency. The loss in performance due to transfer learning was regained by training the network in a real environment for the last few layers only. This resulted in reducing the latency and energy consumption by 1.8 and 3.7 times respectively.

Research in chapter 4 is motivated by how using multi-task RL algorithm results in an energy-efficient system. Such multi-task distributed systems are vulnerable to adversarial attacks. These adversarial attacks can result in a corrupted network with reduced performance for each task. The effect of such an adversary is discussed for various attack methods and an attack method is proposed that outperforms other methods. The proposed attack method is designed to take into account the federated averaging nature at the server by losing all the information gained by other agents/tasks. The effect of these attack methods is used as a motivation for designing a secure multi-task RL algorithm that is immune to such attacks. The proposed Communication Adaptive Federated RL (`ComA-FedRL`) algorithm addresses the adversarial attacks on a Federated RL algorithm. Instead of communicating the policy parameter from all agents at a fixed communication interval, different communication intervals are assigned to agents based on the confidence of them being an adversary. An agent, with higher confidence of being an adversary, is assigned a large communication interval and vice-versa. Communicating less frequently with an adversary agent can greatly mitigate its effects on the learned unified policy. Results on the simple tabular-based RL problem of GridWorld and more complex neural network-based RL problem of AutoNav show that the proposed algorithm outperforms the vanilla algorithm in mitigating the effect

of adversaries.

In chapter 5 we present a modular, end-to-end simulation framework to find a power-performance optimized solution for PIM-based architectures for a given application. The simulation framework encompasses multiple levels of hierarchies including device bitcell, array, memory hierarchy, dataflow, data re-use, and algorithm-to-system mapping. Novel concepts at two levels of the hierarchy are introduced and evaluated: 1. Logic embeddable, high Ion/Ioff Magnetic Tunnel Junction (MTJ) bitcell and 2. Cycle accurate inter and intralayer pipelined operation for high performance and low power operations. Results are compared to pure digital custom ASIC implementation showing orders of magnitude improvements in power-performance on widely accepted MLPerf benchmarks.

Finally, in chapter 6 we present an open-source programming engine for drone-related applications. The motivation behind the tool is to establish a benchmark to implement ML solutions for drones in a standardized set of 3D simulated environments.

Appendices

APPENDIX A

ADDRESSING MULTI-AGENT SYSTEM - MTRL

A.1 Training details

Policy gradient methods for RL is used to train both the GridWorld and AutoNav RL problems. For `ComA-FedRL`, we use a base communication $base\_comm$. In the pre-train phase, the communication interval for each agent is assigned this base communication i.e.

$$comm[i] = base\_comm \quad \forall i \in \{0, n-1\}$$

This means that in the pre-train phase, the agents learn only on local data, and after every $base\_comm$ number of episodes, the locally learned policies are shared with the server for cross-evaluation. This cross-evaluation runs n policies, each on a randomly selected environment and the cumulative reward is recorded. We also take into account the fact that the adversarial agent can present a secondary attack in terms of faking the cumulative reward that it return when evaluating a policy. In the ComA-FedRL implementation, we assume that the adversarial agent returns a cumulative reward of -1, meaning that it fakes the policy being evaluated as adversarial.

At the end of the pre-train phase, the cross evaluated rewards are used to assign communication intervals to all the agents. There are various choices for the selection of this mapping. The underlying goal is to assign a higher communication interval for agents whose policy performs poorly when cross-evaluated and vice versa. We use the mapping shown in Alg. algorithm 6. A reward threshold r_{th} is used to assign agents different communication intervals. If the cumulative reward of a policy in an environment is below r_{th}, it is assigned a high communication interval of $high\_comm$ episodes (marked as a possible adversary), otherwise it is assigned a low communication interval of low_comm episodes

Algorithm 6: Update Communication Intervals

Function `UpdateCommInt(`$r_{m\times n}$, $comm$`)`**:**

Initialize $low\_comm,\ high\_comm,\ r_{th}$

for *each agent i* **do**

Average the rewards across episodes

$$r_{avg} \leftarrow \frac{1}{m}\sum_{j=0}^{m-1} r[:, i]$$

if $r_{avg} \geq r_{th}$ **then**

$comm[i] = low\_comm$

end

else if $r_{avg} < r_{th}$ **then**

if $comm[i] \neq low\_comm$ **then**

$comm[i] = 2 * comm[i]$

end

else

$comm[i] = high\_comm$

end

end

end

return $comm$

(marked as a possible non-adversary). The assigned communication interval also depends on the one-step history of communication intervals. If an agent was previously assigned a higher communication interval and is again marked as a possible adversary, the communication interval assigned to such an agent is doubled. The complete list of hyperparameters used for GridWorld and AutoNav can be seen in Table A.1.

HyperParameter	GridWorld	AutoNav
Functional Mapping	Tabular	Neural Network
Number of agents	4, 8, 12	4
Algorithm	REINFORCE	REINFORCE
Max Episodes	1000	4000
Gamma	0.95	0.99
Learning rate	Variable	1e-4
$base\_comm$	8	8
$wait\_train$	600	1000
Gradient clipping norm	None	0.1
Optimizer type	ADAM	ADAM
Entropy Regularizer Scalar	None	0.5
Training Workstation	GTX1080	GTX1080
Training Time	9 hours	35 hours

Table A.1: Training hyper-parameters for GridWorld and AutoNav

REFERENCES

[1] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, *et al.*, "Recent advances in deep learning for speech research at microsoft," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 2013, pp. 8604–8608.

[2] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[5] H. A. Pierson and M. S. Gashler, "Deep learning in robotics: A review of recent research," *Advanced Robotics*, vol. 31, no. 16, pp. 821–835, 2017.

[6] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis, "Machine learning applications in cancer prognosis and prediction," *Computational and structural biotechnology journal*, vol. 13, pp. 8–17, 2015.

[7] G. Davis, "2020: Life with 50 billion connected devices," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, IEEE, 2018, pp. 1–1.

[8] J. Jin, A. Dundar, and E. Culurciello, "Flattened convolutional neural networks for feedforward acceleration," *arXiv preprint arXiv:1412.5474*, 2014.

[9] M. Wang, B. Liu, and H. Foroosh, "Factorized convolutional neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 545–553.

[10] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[11] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in neural information processing systems*, 2014, pp. 1269–1277.

[12] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[13] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[14] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *arXiv preprint arXiv:1608.03665*, 2016.

[15] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[16] D. Soudry, I. Hubara, and R. Meir, "Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights.," in *NIPS*, vol. 1, 2014, p. 2.

[17] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, Springer, 2016, pp. 525–542.

[18] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[19] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[20] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[21] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and less than 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[22] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[23] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual*

IEEE /ACM International Symposium on Microarchitecture, IEEE Press, 2016, p. 20.

[24] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[25] S. Zeng, Y. Lin, S. Liang, J. Kang, D. Xie, Y. Shan, S. Han, Y. Wang, and H. Yang, "A fine-grained sparse accelerator for multi-precision dnn," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2019, pp. 185–185.

[26] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator," *arXiv preprint arXiv:1811.02883*, 2018.

[27] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[28] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 609–622.

[29] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint:1806.05794*, 2018.

[30] B. Obradovic, T. Rakshit, R. Hatcher, J. A. Kittl, and M. S. Rodder, "High-accuracy inference in neuromorphic circuits using hardware-aware training," *arXiv preprint arXiv:1809.04982*, 2018.

[31] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5687–5695.

[32] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, IEEE, 2017, pp. 1–6.

[33] Y. K. Kwag and J. W. Kang, "Obstacle awareness and collision avoidance radar sensor system for low-altitude flying smart uav," in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, IEEE, vol. 2, 2004, pp. 12–D.

[34] A. S. L. Raimundo *et al.*, "Autonomous obstacle collision avoidance system for uavs in rescue operations," Ph.D. dissertation, 2016.

[35] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[36] Y. Bengio, I. J. Goodfellow, and A. Courville, "Deep learning, book in preparation for mit press (2015)," *Disponivel em http://www. iro. umontreal. ca/bengioy/dlbook*, 2015.

[37] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, *et al.*, "Deep q-learning from demonstrations," *arXiv preprint arXiv:1704.03732*, 2017.

[38] F. Sadeghi and S. Levine, "Cad2rl: Real single-image flight without a single real image," *arXiv preprint arXiv:1611.04201*, 2016.

[39] K. Amer, M. Samy, M. Shaker, and M. ElHelw, "Deep convolutional neural network-based autonomous drone navigation," *arXiv preprint arXiv:1905.01657*, 2019.

[40] D. O. Sales, P. Shinzato, G. Pessin, D. F. Wolf, and F. S. Osorio, "Vision-based autonomous navigation system using ann and fsm control," in *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*, IEEE, 2010, pp. 85–90.

[41] R. Huang, P. Tan, and B. M. Chen, "Monocular vision-based autonomous navigation system on a toy quadcopter in unknown environments," in *Unmanned Aircraft Systems (ICUAS), 2015 International Conference on*, IEEE, 2015, pp. 1260–1269.

[42] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *arXiv preprint arXiv:1705.06963*, 2017.

[43] C. Richter and N. Roy, "Safe visual navigation via deep learning and novelty detection," in *Proc. of the Robotics: Science and Systems Conference*, 2017.

[44] L. Tai, S. Li, and M. Liu, "Autonomous exploration of mobile robots through deep neural networks," *International Journal of Advanced Robotic Systems*, vol. 14, no. 4, p. 1 729 881 417 703 571, 2017.

[45] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," *arXiv preprint arXiv:1704.05588*, 2017.

[46] D. K. Kim and T. Chen, "Deep neural network for real-time autonomous indoor navigation," *arXiv preprint arXiv:1511.04668*, 2015.

[47] A. Giusti, J. Guzzi, D. C. Cireşan, F.-L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, *et al.*, "A machine learning approach to visual perception of forest trails for mobile robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 661–667, 2016.

[48] S. Ross, N. Melik-Barkhudarov, K. S. Shankar, A. Wendel, D. Dey, J. A. Bagnell, and M. Hebert, "Learning monocular reactive uav control in cluttered natural environments," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, IEEE, 2013, pp. 1765–1772.

[49] A. Saxena, S. H. Chung, and A. Y. Ng, "3-d depth reconstruction from a single still image," *International journal of computer vision*, vol. 76, no. 1, pp. 53–69, 2008.

[50] I. Laina, C. Rupprecht, V. Belagiannis, F. Tombari, and N. Navab, "Deeper depth prediction with fully convolutional residual networks," in *3D Vision (3DV), 2016 Fourth International Conference on*, IEEE, 2016, pp. 239–248.

[51] C. Godard, O. Mac Aodha, and G. J. Brostow, "Unsupervised monocular depth estimation with left-right consistency," in *CVPR*, vol. 2, 2017, p. 7.

[52] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[54] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[55] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2017, pp. 786–799.

[56] H. Rebecq, T. Horstschaefer, and D. Scaramuzza, "Real-time visual-inertial odometry for event cameras using keyframe-based nonlinear optimization," in *British Machine Vis. Conf.(BMVC)*, vol. 3, 2017.

[57] D. Palossi, A. Loquercio, F. Conti, E. Flamand, and D. Scaramuzza, "A 64mw dnn-based visual navigation engine for autonomous nano-drones," 2019.

[58] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing dnn computation with relaxed graph substitutions,"

[59] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1633–1685, 2009.

[60] F. L. Da Silva and A. H. R. Costa, "Transfer learning for multiagent reinforcement learning systems.," in *IJCAI*, 2016, pp. 3982–3983.

[61] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[62] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, no. 1, p. 9, 2016.

[63] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," in *International Conference on Artificial Neural Networks*, Springer, 2018, pp. 270–279.

[64] D. George, H. Shen, and E. Huerta, "Deep transfer learning: A new deep learning glitch classification method for advanced ligo," *arXiv preprint arXiv:1706.07446*, 2017.

[65] M. E. Taylor and P. Stone, "Cross-domain transfer for reinforcement learning," in *Proceedings of the 24th international conference on Machine learning*, ACM, 2007, pp. 879–886.

[66] Y. Du, "Improving deep reinforcement learning via transfer," in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 2405–2407.

[67] "Https://www.unrealengine.com/en-us/,"

[68] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and service robotics*, Springer, 2018, pp. 621–635.

[69] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[70] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[71] "Https://developer.nvidia.com/nvidia-system-management-interface,"

[72] "Https://developer.nvidia.com/nvidia-visual-profiler,"

[73] M. A. Anwar and A. Raychowdhury, "Navren-rl: Learning to fly in real environment via end-to-end deep reinforcement learning using monocular images," in *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, IEEE, 2018, pp. 1–6.

[74] I. Yoon, M. A. Anwar, R. V. Joshi, T. Rakshit, and A. Raychowdhury, "Hierarchical memory system with stt-mram and sram to support transfer and real-time reinforcement learning in autonomous drones," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[75] I. Yoon, A. Anwar, T. Rakshit, and A. Raychowdhury, "Transfer and online reinforcement learning in stt-mram based embedded systems for autonomous drones," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1489–1494.

[76] D. Carlo, T. Davide, B. Andrea, and R. Marcello, *Sharing knowledge in multi-task deep reinforcement learning*.

[77] S. Kumar, P. Shah, D. Hakkani-Tur, and L. Heck, "Federated control with hierarchical multi-agent deep reinforcement learning," *arXiv preprint arXiv:1712.08266*, 2017.

[78] H. H. Zhuo, W. Feng, Q. Xu, Q. Yang, and Y. Lin, "Federated reinforcement learning," *arXiv preprint arXiv:1901.08277*, 2019.

[79] G. Palmer, K. Tuyls, D. Bloembergen, and R. Savani, "Lenient multi-agent deep reinforcement learning," *arXiv preprint arXiv:1707.04402*, 2017.

[80] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "A survey and critique of multiagent deep reinforcement learning," *Autonomous Agents and Multi-Agent Systems*, vol. 33, no. 6, pp. 750–797, 2019.

[81] H.-K. Lim, J.-B. Kim, J.-S. Heo, and Y.-H. Han, "Federated reinforcement learning for training control policies on multiple iot devices," *Sensors*, vol. 20, no. 5, p. 1359, 2020.

[82] B. Liu, L. Wang, and M. Liu, "Lifelong federated reinforcement learning: A learning architecture for navigation in cloud robotic systems," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4555–4562, 2019.

[83] S. Zeng, A. Anwar, T. Doan, J. Romberg, and A. Raychowdhury, "A decentralized policy gradient approach to multi-task reinforcement learning," *arXiv preprint arXiv:2006.04338*, 2020.

[84] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, *et al.*, "Towards federated learning at scale: System design," *arXiv preprint arXiv:1902.01046*, 2019.

[85] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[86] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv:1610.02527*, 2016.

[87] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[88] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[89] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *arXiv preprint arXiv:1611.01236*, 2016.

[90] F. Tramér, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," *arXiv:1705.07204*, 2017.

[91] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[92] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.

[93] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European symposium on security and privacy (EuroS&P)*, IEEE, 2016, pp. 372–387.

[94] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, "Adversarial attacks on neural network policies," *arXiv preprint arXiv:1702.02284*, 2017.

[95] J. Kos and D. Song, "Delving into adversarial attacks on deep policies," *arXiv preprint arXiv:1705.06452*, 2017.

[96] Y. Huang and Q. Zhu, "Deceptive reinforcement learning under adversarial manipulations on cost signals," in *International Conference on Decision and Game Theory for Security*, Springer, 2019, pp. 217–237.

[97] Y. Ma, X. Zhang, W. Sun, and J. Zhu, "Policy poisoning in batch reinforcement learning and control," in *Advances in Neural Information Processing Systems*, 2019, pp. 14 570–14 580.

[98] Y.-C. Lin, Z.-W. Hong, Y.-H. Liao, M.-L. Shih, M.-Y. Liu, and M. Sun, "Tactics of adversarial attack on deep reinforcement learning agents," *arXiv preprint arXiv:1703.06748*, 2017.

[99] V. Behzadan and A. Munir, "The faults in our pi stars: Security issues and open challenges in deep reinforcement learning," *arXiv preprint arXiv:1810.10369*, 2018.

[100] A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell, "Adversarial policies: Attacking deep reinforcement learning," *arXiv preprint arXiv:1905.10615*, 2019.

[101] P. Blanchard, R. Guerraoui, J. Stainer, *et al.*, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems*, 2017, pp. 119–129.

[102] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, "Analyzing federated learning through an adversarial lens," in *International Conference on Machine Learning*, PMLR, 2019, pp. 634–643.

[103] C. Xie, Y. Wu, L. v. d. Maaten, A. L. Yuille, and K. He, "Feature denoising for improving adversarial robustness," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 501–509.

[104] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," *arXiv preprint arXiv:1703.02702*, 2017.

[105] A. Mandlekar, Y. Zhu, A. Garg, L. Fei-Fei, and S. Savarese, "Adversarially robust policy learning: Active construction of physically-plausible perturbations," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 3932–3939.

[106] A. Pattanaik, Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary, "Robust deep reinforcement learning with adversarial attacks," *arXiv preprint arXiv:1712.03632*, 2017.

[107] C. Tessler, Y. Efroni, and S. Mannor, "Action robust reinforcement learning and applications in continuous control," *arXiv preprint arXiv:1901.09184*, 2019.

[108] N. Rodríguez-Barroso, E. Martínez-Cámara, M. Luzón, G. G. Seco, M. Á. Veganzones, and F. Herrera, "Dynamic federated learning model for identifying adversarial clients," *arXiv preprint arXiv:2007.15030*, 2020.

[109] R. R. Yager and D. P. Filev, "Induced ordered weighted averaging operators," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 2, pp. 141–150, 1999.

[110] A. Anwar and A. Raychowdhury, "Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning," *IEEE Access*, vol. 8, pp. 26 549–26 560, 2020.

[111] Y. K. Lee, Y. Song, J. Kim, S. Oh, B.-J. Bae, S. Lee, J. Lee, U. Pi, B. Seo, H. Jung, *et al.*, "Embedded stt-mram in 28-nm fdsoi logic process for industrial mcu/iot application," in *2018 IEEE Symposium on VLSI Technology*, IEEE, 2018, pp. 181–182.

[112] Z. Jiang, Y. Wu, S. Yu, L. Yang, K. Song, Z. Karim, and H.-S. P. Wong, "A compact model for metal–oxide resistive random access memory with experiment verification," *IEEE Transactions on Electron Devices*, vol. 63, no. 5, pp. 1884–1892, 2016.

[113] Y. L. et al, "A variation robust dnn inference engine based on stt-mram with parallel read-out," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2013, pp. 1–8.

[114] H. Noguchi, K. Kushida, K. Ikegami, K. Abe, E. Kitagawa, S. Kashiwada, C. Kamata, A. Kawasumi, H. Hara, and S. Fujita, "A 250-mhz 256b-i/o 1-mb stt-mram with advanced perpendicular mtj based dual cell for nonvolatile magnetic caches to reduce active power of processors," in *2013 Symposium on VLSI Technology*, IEEE, 2013, pp. C108–C109.

[115] K. R. Shubham Jain Abhronil Sengupta and A. Raghunathan, "Rxnn: A framework for evaluating deep neural networks on resistive crossbars," *arXiv preprint arXiv:1809.00072*, 2019.

[116] X. Peng, R. Liu, and S. Yu, "Optimizing weight mapping and data flow for convolutional neural networks on rram based processing-in-memory architecture," in

2019 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2019, pp. 1–5.

[117] "Https://github.com/neurosim/dnn_neurosim_v1.0,"

[118] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, p. 61, 2015.

[119] Y. Kim, Y. Zhang, and P. Li, "A digital neuromorphic vlsi architecture with memristor crossbar synaptic array for machine learning," in *2012 IEEE International SOC Conference*, IEEE, 2012, pp. 328–333.

[120] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, *et al.*, "Reno: A high-efficient reconfigurable neuromorphic computing accelerator design," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2015, pp. 1–6.

[121] C. Yakopcic and T. M. Taha, "Energy efficient perceptron pattern recognition using segmented memristor crossbar arrays," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2013, pp. 1–8.

[122] T. M. Taha, R. Hasan, C. Yakopcic, and M. R. McLean, "Exploring the design space of specialized multicore neural processors," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2013, pp. 1–8.

[123] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2007, pp. 3–14.

VITA

Aqeel Anwar received the Bachelor's degree in Electrical Engineering from the University of Engineering and Technology (UET), Lahore, Pakistan, in 2012. He obtained his Master's degree in Electrical and Computer Engineering from the Georgia Institute of Technology, Atlanta, Georgia, USA, in 2017, as a recipient of the Fulbright Scholarship. Currently, he is pursuing a Ph.D. degree in Electrical and Computer Engineering from the Georgia Institute of Technology Atlanta Georgia USA under the supervision of Dr. Arijit Raychowdhury. He is working towards shifting machine learning (ML) from cloud to edge nodes by improving the energy efficiency of current state-of-the-art ML algorithms. His research focuses on modifying existing ML algorithms, designing energy-efficient ML accelerators, and leveraging the hardware-algorithm co-design to shift ML from server/ cloud to compute-limited edge nodes.

He is the recipient of the prestigious Fulbright Fellowship for the Masters's program at Georgia Institute of Technology. He was awarded the Best Student Paper Award at the 25^{th} International Conference on Mechatronics and Machine Vision in Practice (M2VIP) held $20 - 22^{nd}$ November 2018 in Germany for his paper entitled "NavREn-Rl: Learning to fly in real environment via end-to-end deep reinforcement learning using monocular images". Apart from academia, he has had industry experience at various levels. He has worked with a consulting software house, a research organization, and an autonomous vehicle startup. He has worked as an intern at Samsung Semiconductor Inc. in the summer of 2019 where he published two patents on energy-efficient ML accelerator.

He has actively participated in various technical and cultural societies. He was the vice president of the Institution of Engineering and Technology (IET) university chapter, a UK-based Engineering society during his undergraduate studies. At Georgia Tech, he has been the senator for the Student Government Association and the Vice President of the Pakistan

Student Association. Apart from research, he likes running and playing badminton. His hobbies include making 3D paper models of various buildings and skyscrapers around the world. He likes traveling and has been to 3 continents, over 15 countries, and 25 US states. In his free time, he writes articles on Machine Learning and Ph.D. life lessons.